RL-TR-97-205
Final Technical Report
October 1997

CONCEPT OF OPERATIONS FOR A VIRTUAL MACHINE FOR C3I APPLICATIONS

Bagrodia and Chandy Associates

Rajive Bagrodia and Mani Chandy

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Rome, New York

APPROVED: [illegible]

PAUL M. ENGELHART
Project Engineer

FOR THE DIRECTOR:

WARREN H. DEBANY, JR.
Technical Advisor
Command, Control & Communications Directorate

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE October 1997	3. REPORT TYPE AND DATES COVERED Final Feb 95 - Jan 96

4. TITLE AND SUBTITLE CONCEPT OF OPERATIONS FOR A VIRTUAL MACHINE FOR C3I APPLICATIONS	5. FUNDING NUMBERS C - F30602-95-C-0045 PE - 62702F PR - 5581 TA - 18 WU - PK
6. AUTHOR(S) Rajive Bagrodia and Mani Chandy	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Bagrodia and Chandy Associates 10495 Colina Way Los Angeles CA 90077	8. PERFORMING ORGANIZATION REPORT NUMBER N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory/C3CB 525 Brooks Road Rome NY 13441-4505	10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-97-205

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Paul M. Engelhart/C3CB/(315) 330-4477

12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This final technical report summarizes research accomplished by Bagrodia and Chandy Associates. This 12-month research endeavor, entitled "Concept of Operations for a Virtual Machine for C3I Applications," examined issues in using a concurrent virtual machine for the design of C3I applications, where a concurrent virtual machine is an abstraction of a parallel machine. These issues were evaluated and defined for the purposes of handling concurrency in the design of parallel programs and to facilitate porting a parallel program to multiple architectures. This report provides a description of requirements analysis, state-of-the-art review, technology trends, concurrent virtual machine, cost benefit analysis and case study undertaken. One of the recommendations proposed demonstrates the applicability of the proposed concurrent virtual machine for the design of a terrain masking program.

14. SUBJECT TERMS Virtual Machine, Parallel Software Development, C3I Systems, High Performance Computing	15. NUMBER OF PAGES 134
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

TABLE OF CONTENTS

LIST OF FIGURES

OVERVIEW

This report evaluates the use of a concurrent virtual machine (CVM) for the design of C^3I applications. A CVM is an abstraction of a parallel machine. It is defined for the purposes of handling concurrency in the design of parallel programs and to facilitate porting a parallel program to multiple architectures. The study had the following primary tasks:

- **requirements analysis:** identify the critical characteristics of C^3I systems and the key differences between military C^3I systems and other command and control systems.
- **state of the art:** review state of the art in design methodologies for parallel C^3I applications. The design of a C^3I system entails looking at parallelism at multiple levels beginning with the architecture level, to operating systems, programming languages, libraries, objects or similar composable approaches, specification notations, to the Problem Solving Environments (PSEs) at the application level. A detailed review of the state of the art was done at each of these levels, to examine their impact on CVMs for C^3I applications. The preceding layers form a hierarchy, and a CVM can be developed at each of these layers. When porting an application from one machine to another, it is necessary to consider the impact at each of these layers and suggest appropriate methods and tools in the CVM.
- **technology trends**: evaluate current technology trends in the industry and identify their potential impact on the design of CVMs.
- **concurrent virtual machine**: based on the requirements analysis for a CVM and the technology trends, the study proposed a hierarchy of CVMs for C^3I applications. The goals for the CVM were outlined; the architecture, application, and user space for the CVM was defined, and a programming model was proposed.
- **cost benefit analysis:** the primary benefits of using a CVM for the design of C^3I applications were identified together with the costs that must be incurred for such a methodology.
- **case study:** The virtual machine based design methodology that was proposed in this study was applied to the design of a program for terrain masking. This application is one among a suite of C^3I benchmarks called the C^3I Parallel Benchmark Suite (C3IPBS) that was developed under another Rome Laboratory initiative.

The rest of this document is organized as follows: we enclose a summary of our recommendations for the CVM hierarchy for C^3I applications and detail each of the layers in the hierarchy. The recommendations also demonstrate the applicability of the proposed CVM for the design of a terrain masking program.
Three appendices provide additional detail: Appendix 1 describes the key characteristics of C^3I applications and describes the architecture, application, and user space considered in this study. Appendix 2 provides a cost benefit analysis of the use of concurrent virtual machines and is a detailed discussion of the state of the art and the design issues for virtual machines at each of the following levels of the hierarchy: architecture, operating system, programming languages, high-level notations, interoperable objects, and application levels. Appendix 3 is a detailed report on the design of programs for the terrain masking problem using the proposed CVM hierarchy. The section describes the

problem, proposes a design using a high-level notation, and designs a set of programs using different programming paradigms using the CVM methodology.

OVERALL RECOMMENDATIONS

***Recommendation*:**

Adopt CVM Hierarchy

- Problem-Solving Environment that operates on a hierarchy of CVMs is feasible, and its cost/benefit ratio appears favorable.
- The suggestion for the hierarchy is based on observations about converging technologies, coupled with the need to remain flexible.

Different virtual machines at different levels of design

The concept of a virtual machine is useful at several levels of design, from hardware to the application. Virtual machines at different levels of design are concerned with different levels of detail. For example, a virtual machine for parallel hardware is different from a virtual machine for C3I applications.

Hierarchy of virtual machines.

We recommend that Rome Laboratory base its C3I designs on a hierarchy of virtual machines, with different virtual machines appropriate for different levels in the design. We recommend against adopting a single virtual machine for all levels of design because we believe that designers will have to participate in mapping an application on a virtual machine at one level on to a virtual machine at a lower level. Automatic mapping from an application-level virtual machine to a real parallel machine may not give desired performance. A hierarchy of virtual machines allows for automatic mapping, and also allows designers to direct the mapping.

Recommendation: A specific virtual machine hierarchy

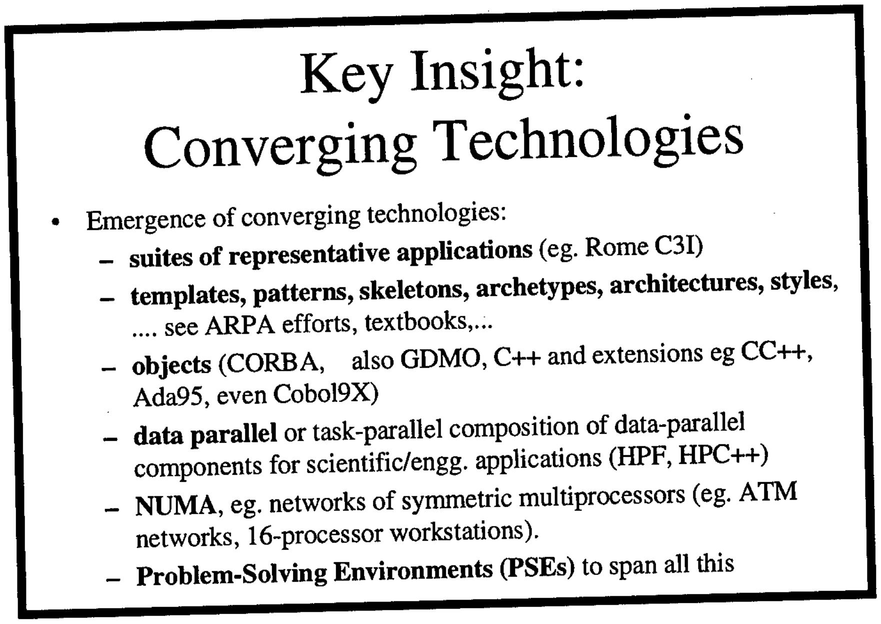
We recommend a specific virtual machine hierarchy. Our recommendation is based on the observation that technologies that impact virtual machines are converging. Therefore, a plethora of virtual machines is not required. The virtual-machine hierarchy must, however, be flexible to handle evolving technologies.

Layers of the virtual machine hierarchy

The hierarchy has the following layers:

1. **application layer**
2. **software architecture layer:** composition of objects that provide different functionality,
3. **programming model**: the model of parallel computation within an object that provides a single functionality, and
4. **hardware layer**: model of the parallel hardware.

These layers are described later. We recommend a specific virtual machine hierarchy because there is a convergence of technologies at each level of the hierarchy; this convergence is described next.



Key Insight: Converging Technologies

- Emergence of converging technologies:
  - **suites of representative applications** (eg. Rome C3I)
  - **templates, patterns, skeletons, archetypes, architectures, styles**, ... see ARPA efforts, textbooks,...
  - **objects** (CORBA, also GDMO, C++ and extensions eg CC++, Ada95, even Cobol9X)
  - **data parallel** or task-parallel composition of data-parallel components for scientific/engg. applications (HPF, HPC++)
  - **NUMA**, eg. networks of symmetric multiprocessors (eg. ATM networks, 16-processor workstations).
  - **Problem-Solving Environments (PSEs)** to span all this

We have identified several technologies that we use in the virtual machine hierarchy. They are:

1. **The application level --- Defining the application space**: suites of representative applications are employed to describe an application space;
2. **The software architecture level --- Reuse of design patterns and implementation**: Methods such as templates, patterns, archetypes and software architectures for developing the design infrastructure to support continuous evolution of families of related products;
3. **Composing elements with state (memory):** object composition technologies such as CORBA and OLE;

4. **A parallel programming model**: a consensus programming model based on task-parallel composition of data-parallel components;
5. **Hardware level**: a consensus architecture model based on networks of symmetric multiprocessors; and,
6. **Integration of all levels through a problem-solving environment (PSE):** PSEs that deal with these technologies in a comprehensive way.

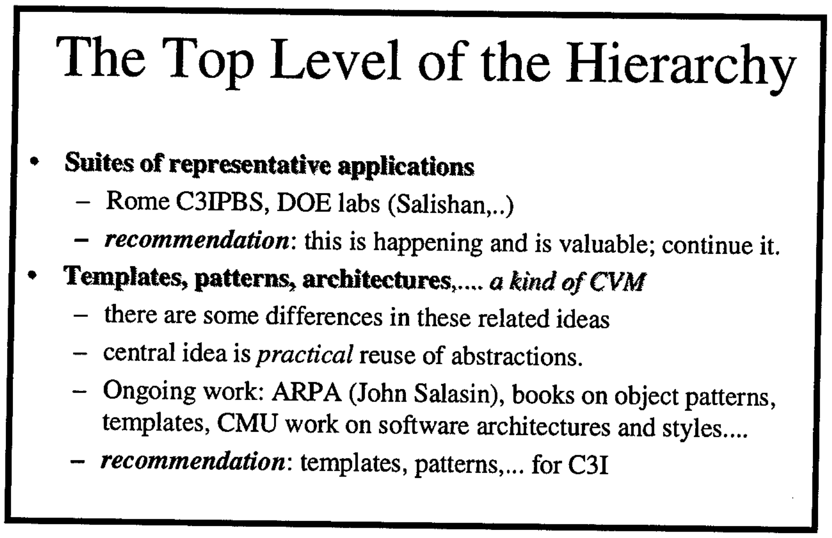
Each of these technologies deals with different aspects of virtual machines.

1. The application layer.

What is the relevant application space?

The first step in proposing a virtual machine, or a collection of virtual machines, is to define the broad class of applications of interest. Defining a virtual machine for all applications is not possible because the range of applications is too diverse. Therefore, a central question is: How is the application space described?

We recommend describing the application space by a suite of representative applications. Applications of interest are applications similar to those in the suite. Rome Laboratory has developed a suite of C3I applications called the C3IPBS. Though this suite was not designed for virtual machine design, it serves that purpose. Thus, the answer to the question, "What is the application space?" is "Applications such as those in C3IPBS." More applications can be added to the suite later; the important point for now is that virtual machine designers have a way of understanding the space of relevant applications.



The Top Level of the Hierarchy

- **Suites of representative applications**
  - Rome C3IPBS, DOE labs (Salishan,..)
  - ***recommendation***: this is happening and is valuable; continue it.
- **Templates, patterns, architectures,....** ***a kind of CVM***
  - there are some differences in these related ideas
  - central idea is *practical* reuse of abstractions.
  - Ongoing work: ARPA (John Salasin), books on object patterns, templates, CMU work on software architectures and styles....
  - ***recommendation***: templates, patterns,... for C3I

2. The Software Architecture Layer

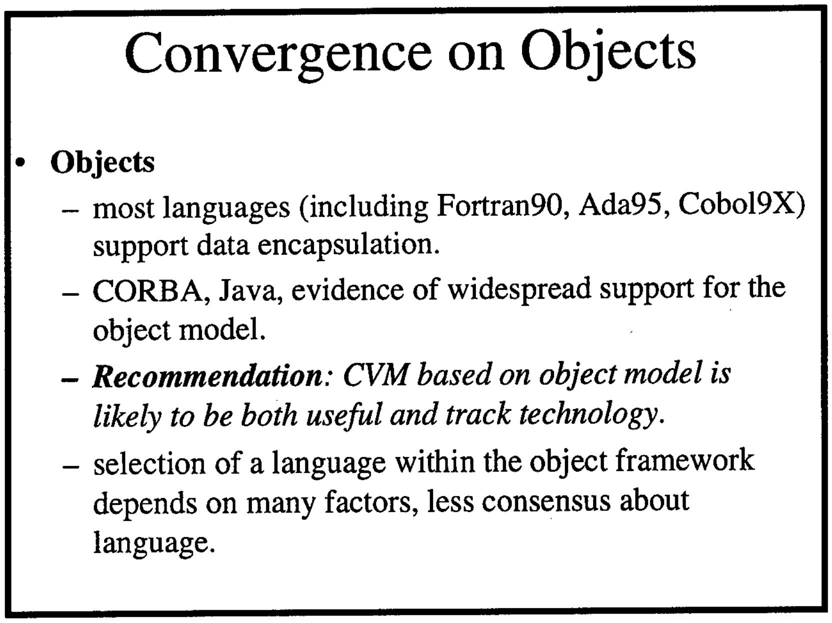
Virtual machines targeted at the application space.

Virtual machines can be targeted at different levels in design, from hardware to application. This paragraph deals with virtual machines at the application level. Virtual machines provide the functions required by the application space; the more complete the functions provided by the virtual machine, the simpler the problem of designing a particular application.

Software development groups do not develop a succession of independent pieces of software; they continually modify and refine a family of related products. An application-level virtual machine is the software infrastructure that supports refinement of a product family. An application-level virtual machine for C3I is the software infrastructure that supports the development of applications such as those in C3IPBS. The virtual machine is a good one if it allows designers to reuse functionality common to most of the applications in the product family.

The concept of a virtual machine for a family of related, continuously-evolving software products is related to other concepts including software architectures and architectural styles, patterns of object-oriented designs, functional skeletons, and parallel archetypes. There is a convergence on the basic concepts of application-level virtual machines, though the approaches diverge in the details. Rome Laboratory can exploit this convergence by leveraging the research (funded by DOD and other agencies) in this area.

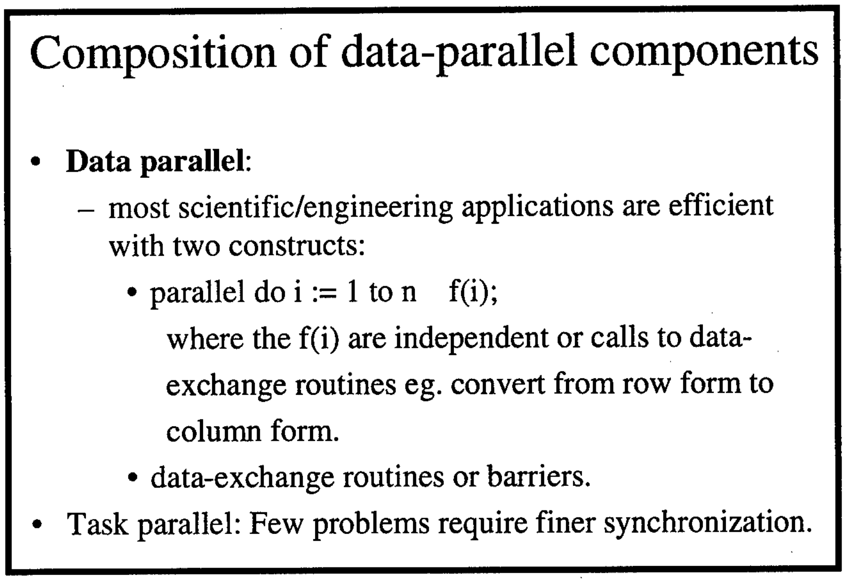
We recommend that the top level of the virtual machine hierarchy be based on the broad class of related concepts that include design patterns, templates, archetypes and software architectures.



Convergence on Objects

- **Objects**
  - most languages (including Fortran90, Ada95, Cobol9X) support data encapsulation.
  - CORBA, Java, evidence of widespread support for the object model.
  - ***Recommendation****: CVM based on object model is likely to be both useful and track technology.*
  - selection of a language within the object framework depends on many factors, less consensus about language.

3. Composing elements with state.
Widespread interest in C++, Java, Smalltalk, ADA95, Cobol9X, CORBA and OLE is indicative of the convergence in object-oriented technologies for program composition. There is, as yet, no consensus on the best object-oriented language, but there is a growing consensus for using objects for dealing with components that have state. A C3I system is (usually) a composition of several components, each of which has some state (local memory). *Our recommendation is that Rome Laboratory use object-composition technologies for integrating state-based components;* the concept of objects will have to be extended to include threads (loci of control) within objects.



Composition of data-parallel components

- **Data parallel**:
  - most scientific/engineering applications are efficient with two constructs:
    - parallel do i := 1 to n   f(i);

      where the f(i) are independent or calls to data-exchange routines eg. convert from row form to column form.
    - data-exchange routines or barriers.
- Task parallel: Few problems require finer synchronization.

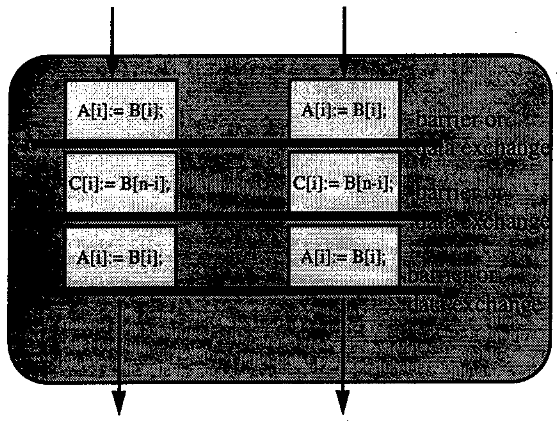
4. The Programming Model

We recommend a programming model which consists of a task-parallel framework, the elements of which are data-parallel components. Next, this programming model is described in greater detail. Our reasons for the recommendation are given later.

Most scientific and engineering applications in use today are based on two constructs:

1. Creation of multiple processes by means of a parbegin/parend or pardo/endpardo, or implicitly by specifying the number of processes. In a shared memory model, threads that share common memory are used in place of processes.
2. Simple synchronization mechanisms based on barriers or data-exchanges between processes. A barrier is a point in the computation at which all processes (or threads) synchronize. A data-exchange is a set of communications that often implements a barrier in addition to exchanging information among processes.

Data Parallel Structure



Data parallel:
typical scientific
and engg.
applications.

only synchronization
is the barrier and
information exchange.

There are, of course, more complex programming models with different synchronization mechanisms such as semaphores and monitors. Also, it is not necessary that groups of processes synchronize at a barrier or a data-exchange; a completely asynchronous model is possible. Nevertheless, group synchronization and group communication are the norm for scientific and engineering applications other than discrete-event simulation. The basic programming model is that of a group of processes or threads executing in semi-synchrony.

The programming model for C3I applications is that of independent processes; pairs of processes communicate with each other in an asynchronous fashion, as opposed to large groups of processes interacting synchronously. A C3I application may use reliable broadcast or multicast; usually, however, all the processes in a C3I application are not identical or even very similar, and all processes do not operate in a synchronous fashion. This model is called the task-parallel or function-parallel model.

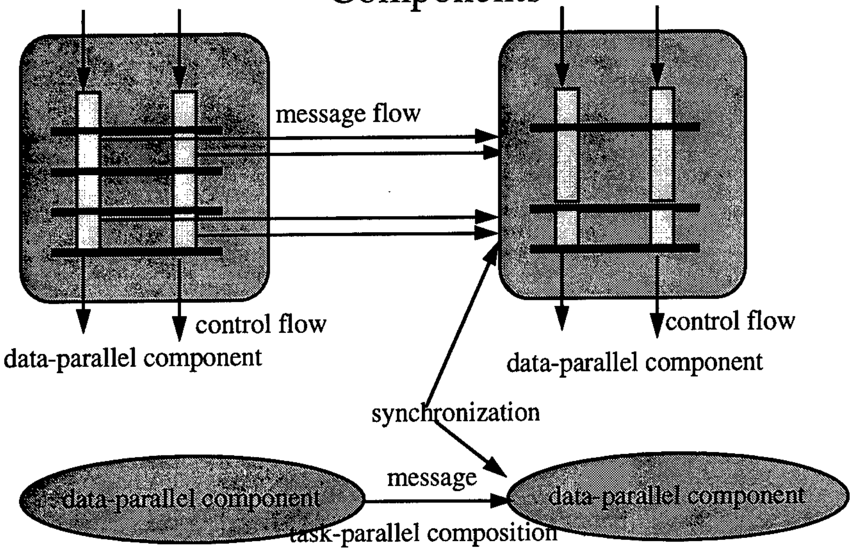
Task-parallelism is necessary as the framework for integrating different processes (sensors, threat-identification, computation of response), and data-parallelism is necessary for the scientific computations carried out within each process. *We recommend a task-parallel framework in which the components of the framework are either sequential programs or data-parallel programs.*

Recommendation: Task-Parallel Composition of Data-Parallel Parts

- **CVM based on task-parallel composition of data-parallel components is flexible.**
- Data-parallel model is straightforward and widely-used.
- Task-parallel composition introduces synchronization other than barrier; this additional synchronization can be difficult for many programmers.
- But, task parallelism is helpful in some cases and necessary in a few.

More complex programming models are possible, such as a data-parallel framework, the elements of which were task-parallel networks. An important programming model is that used in functional programming --- each component of the task-parallel framework could be a functional program rather than a data-parallel program. Even in this case, however, the basic programming model remains unchanged: It is a task-parallel structure with components that can either be sequential (including functional programs) or data-parallel.

Task-Parallel Composition of Data-Parallel Components



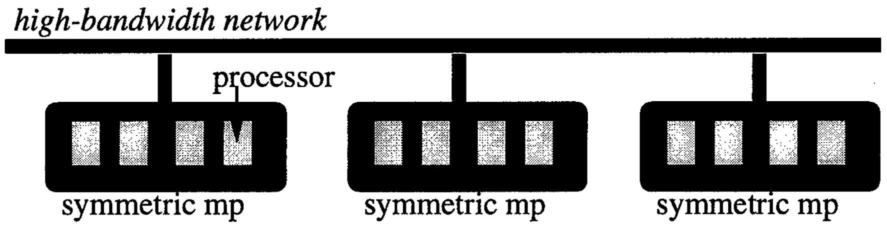
5. The model of hardware

C3I systems have components of many different types. A component can be a parallel computer. The virtual machine at the hardware level deals with both (1) different kinds of components interconnected in arbitrary and possibly dynamic fashion, and (2) some components, in turn, being parallel machines.

A consensus parallel architecture mirrors the composition of different elements, where each element is a collection of identical smaller elements. This architecture is a network of shared-memory multiprocessors. The network can be regular as in the case of the Intel Paragon or the IBM SP machines, or it can be irregular as in the case where the network reflects a C3I application. Each element of the network is a shared-memory multiprocessor such as the Intel P6 microprocessor with some number of symmetric shared-memory multiprocessors.

CVM at Architecture Level

- NUMA (nonuniform memory access)
  - two-level or three-level memory model.
    - cache (ignored for high-level programming)
    - shared-memory symmetric multiprocessing
    - remote access via high-bandwidth network.



The two-level structure of the consensus parallel architecture is also similar to the two-level structure of the programming model. In some cases, only one level of the structure may be used; for instance a computer can be a shared-memory multiprocessor, or it can be a network of uniprocessors.

We recommend a hardware virtual machine that consists of a network of symmetric multiprocessors.

Recommendation: CVM at Architecture Level

- Convergence towards shared address space model for scientific computations.
- Shared address space supported in hardware as in symmetric multiprocessors or logically in distributed shared memory or global/local pointers as in CC++.
- Message-passing required for composing C3I components.
- ***Recommendation***: CVM: NUMA, and specifically, networks of single-address space machines. Consistent with software CVM recommendation.

6. Integration of all levels of the virtual machine hierarchy

We have recommended a specific virtual machine hierarchy.*We also recommend integrating all the virtual machines in the hierarchy by means of a problem-solving environment.* Designers need tools and methods to help them map an application defined in terms of the interface (API) for the top-level virtual machine to the hardware. A

Recommendation: Problem-Solving Environments

- Need methods and tools to map from CVM at the specification pattern level to the CVM at the machine-architecture level. Such an integrated collection of tools is a problem-solving environment (PSE).
- ***Recommendation***: Develop CVM-based PSEs for C3I.

problem-solving environment that is an integrated collection of mapping tools from layers in the virtual machine hierarchy to lower layers helps designers exploit virtual machines. *We recommend the development of problem-solving environments specifically to deal with mapping applications developed at one level of the virtual machine hierarchy to lower levels.*

A Case Study

Next, we present a C3I case study that illustrates the recommended virtual machine hierarchy. The example is*terrain masking*. The terrain masking problem is a problem in the C3IPBS problem suite developed by Rome Laboratory. Thus, it is an example of the use of a problem suite to describe a problem area. We designed virtual machines at each layer of the hierarchy, and we carried out a cost-benefit analysis of the value of virtual machines.

The Problem

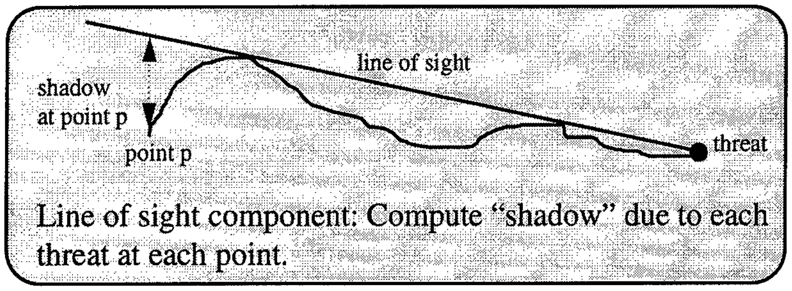
Here we give a brief description of the problem; more detailed descriptions are given elsewhere.

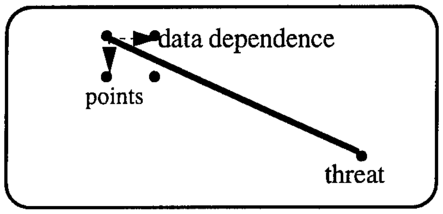
Given:

1. A rectangular grid in a two-dimensional Cartesian space; at each point in the grid we are given the elevation at that point. In other words, we are given a function that maps grid points to height.
2. A set of threats. Each threat is fixed, and its location is a grid point. The locations of the threats are given. We are also given additional information about threats.

A threat can be treated as a source of light. Each threat (light source) casts a shadow. Different threats cast different shadows. The problem is to determine the region that is in darkness.

Terrain Masking Components: Line-of-sight component


line of sight
shadow at point p
point p
threat
Line of sight component: Compute "shadow" due to each threat at each point.



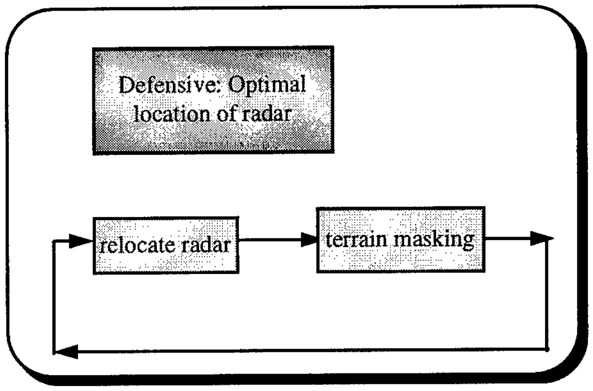
Template captured by data dependence pattern.

Output The output of the program is, for each grid point, the maximum height below which the region is in shadow.

Requirements Analysis: How is Terrain Masking Used?
Before we start designing a solution using the virtual machine hierarchy, we first carry out a requirements analysis to identify how the terrain masking problem is used. A terrain masking application can be used in an offensive or defensive capacity. In the offensive capacity, the problem is given the enemy's threat location compute regions of safety for our airborne devices to penetrate the enemy's space as close to their targets as possible without being visible to the enemy. In a defensive capacity the problem is to locate radar sites so as to deny the enemy invisible 'safe' areas; our defensive radar sites may be on the ground, airborne or in submarines, and they can be static or mobile. The manner in which the application is used influences the design of the parallel application.

Terrain-Masking: Case Study
How the recommendations fit in

- Requirements analysis of applications that can use terrain masking as a component.



Offensive:
*optimal trajectory
*moving threats as in airborne or submarine mobile threats

The use of terrain masking in defensive operations is an optimization problem with the terrain masking application as a component within the overall optimization. An approach to the optimization problem is to iteratively compute locations of defensive radar and then call the terrain masking application, as shown in the figure.

The composition of the terrain masking application within the larger optimization problem is straightforward. Object composition (for instance, by using object request brokers --- ORBS) can be used but is not necessary for this simple application.

Next, we describe how each level of the virtual machine hierarchy is used, starting at the top levels.

The software architecture
Our first step is to partition the application into reusable components. Since parallel programming is more difficult than sequential programming, and since there are fewer programmers experienced in parallel computing, we are particularly concerned with reusing parallel structures to simplify the design task.

We attempt to identify a parallel program archetype that fits a component of the problem. A parallel program archetype is a program structure that deals with process creation, communication and synchronization. A parallel archetype does not deal with sequential programming issues. A parallel archetype is refined for a specific problem by "filling in" the slots of the archetype. In object-oriented terms a parallel archetype is a class that deals with parallelism, and the archetype is tailored to a specific application by providing sequential methods for that application.

If the archetype we seek exists in a library of archetypes then we reuse it.; if it does not exist in a library, then we implement the archetype and insert it into the library. For the terrain masking problem two components with well-defined parallel archetypes suggest themselves.

1. An archetype for line of sight calculations that deal with computing the "shadow" for a light source.
2. An archetype for computing the cumulative effect of a set of light sources.

Next, we describe these archetypes in more detail.

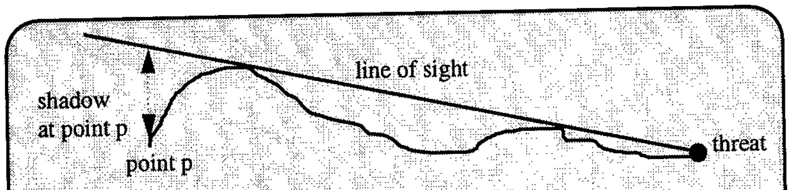
Partition application into reusable components

- Terrain masking:
  - line of sight calculations
  - cumulative effect of threats at each point which is independent for each point

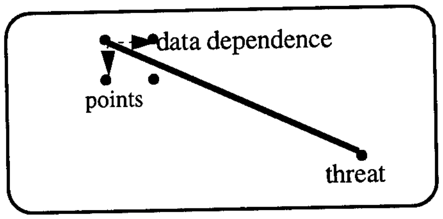
Parallel archetypes defined in terms of dataflow dependency patterns
The dataflow dependency in a line-of-sight calculation has the following form. The height of the shadow at a grid point **p** thrown by a light source depends on values at neighboring grid points between **p** and the light source. This data dependency is shown in the following figure. The form of the data dependency allows many forms of parallel computations, such as parallel computations of values at grid points in concentric rings around the light source, as shown in the figure.

The parallel archetype forlight source calculations is the archetype that deals with this particular form of data dependence. This data dependence structure appears in many other problems including very different problems in matrix algebra. Thus the parallel archetype can be reused, with the programmer only specifying the sequential method at each grid point as a function of the values at neighboring grid points.

Terrain Masking Components: Line-of-sight component



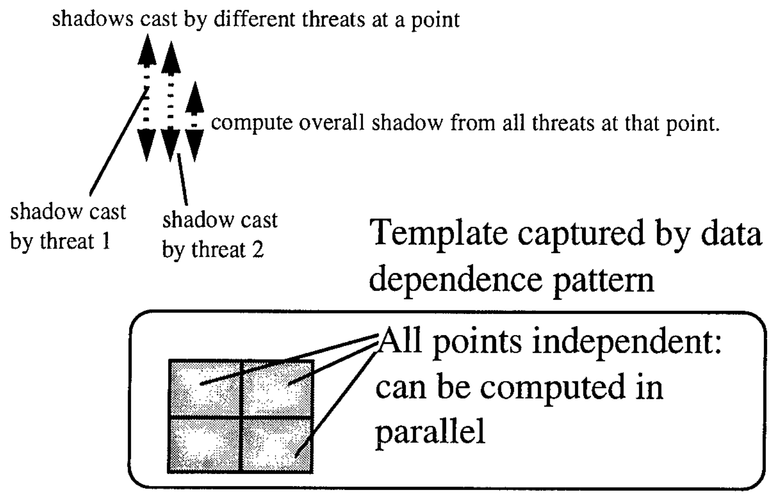
Line of sight component: Compute "shadow" due to each threat at each point.



Template captured by data dependence pattern.

A second parallel archetype deals with computing the aggregate effect of all threats. In this example, the aggregate effect at a grid point is the shadow cast by all light sources; the height of the aggregate shadow is the minimum of the heights of the shadows cast by all the light sources. Thus, the computation at a grid point is entirely local to the grid point. The data dependency that defines this archetype is also extremely common, and here too an application can be developed by providing the*sequential* methods that tailor the archetype to the specific application.

Terrain Masking Components: Computing net effect of all threats


shadows cast by different threats at a point
compute overall shadow from all threats at that point.
shadow cast by threat 1
shadow cast by threat 2
Template captured by data dependence pattern
All points independent: can be computed in parallel

The Virtual Machine at the Application Level: 3D+T+M

We recommend using the following virtual machine at the C3I application level. The problem space is 3-dimensional space, plus T (time), plus another dimension M for models. The 3-D space can be organized in one of a small number of coordinate systems. An API is used to manipulate the 3-D + T + M space. The API is defined, in part, by a set of archetypal methods. In the terrain masking example we use two archetypes. The archetypes are defined by the data dependence pattern. There are a small number of data dependence patterns that arise often in the 3D + T problem space, and an archetype library should handle the more common cases. A program designer uses the API for the 3D+T+M space, focusing on the sequential methods that must be provided to tailor the archetype for a specific application.

This is not the only application-level virtual machine appropriate for C3I. It is a good example of an important C3I virtual machine. A small number of C3I application virtual machines will help simplify design for many C3I problems.

We recommend the development of a 3D+T+M virtual machine, with an API that has GUI support to search through a library of problem space descriptions and a library of archetypes, helps in selection appropriate archetypes and supports tailoring the virtual machine to a specific application by helping in adding the sequential components. A valuable exercise is to compare (a) the productivity and (b) the efficiency of code developed using the virtual machine approach with a traditional approach.

Recommendation

- *Build library of data-dependence patterns on grids, and GUI interface to help users browse through and select patterns.*
- Instantiate a parallel program by filling in slots in the pattern, eg. line-of-sight calculations. The slots are filled in by sequential functions.
- Compose components into a program.
- Compare efficiencies of programs developed using templates with hand-tailored programs.

The Programming Model

The programming model in our virtual machine hierarchy is task parallel composition of data-parallel components. Both the parallel archetypes are mapped on to the programming model. The user of an archetype does not have to deal with the parallel programming model because the archetype user only provides the*sequential* methods required to tailor an archetype to a specific application. Thus, a user at one level of the hierarchy can map automatically down several levels in the hierarchy, all the way to a target parallel machine. There may, however, be cases where an automatic mapping does not have the efficiency required by the application. In such cases, the user can step in and direct the mapping to the next level.

Next, we describe the mapping from the application-level virtual machine to the programming-model virtual machine. This description is in terms of the mapping from both archetypes used in the terrain-masking application to the programming model: task-parallel composition of data-parallel programs.

Archetype 1: Computations emanating from a point in 3-D space.
The first archetype deals with computing the shadow cast by a light source at a point. The problem data set is partitioned among processes or threads in rectangular blocks. The shadow within a rectangular region can be computed when data is available at the two boundaries between the region and the light source. The design issues are the size of the rectangular blocks, and the method of synchronization between blocks.

There are two straightforward methods of synchronization: (1) using barriers, or (2) using messages or signals to inform a process responsible for a rectangular block of data that data for its boundaries is available. Both methods fit into the programming model. In the first case the programming model is data parallel; parallelism is restricted to the initial creation of threads and subsequent barriers. In the second case the programming model is task parallel with each task responsible for its portion of the rectangular boundary.

Archetype 2: Independent local computations.
The same data decomposition --- rectangular blocks --- is considered. The computation of each block is completely independent of other blocks. Thus the computation of all blocks can be considered to be a data-parallel operation.

The two archetypes are composed using sequential composition; when the first archetype computation completes for all light sources, the second archetype computation is initiated.

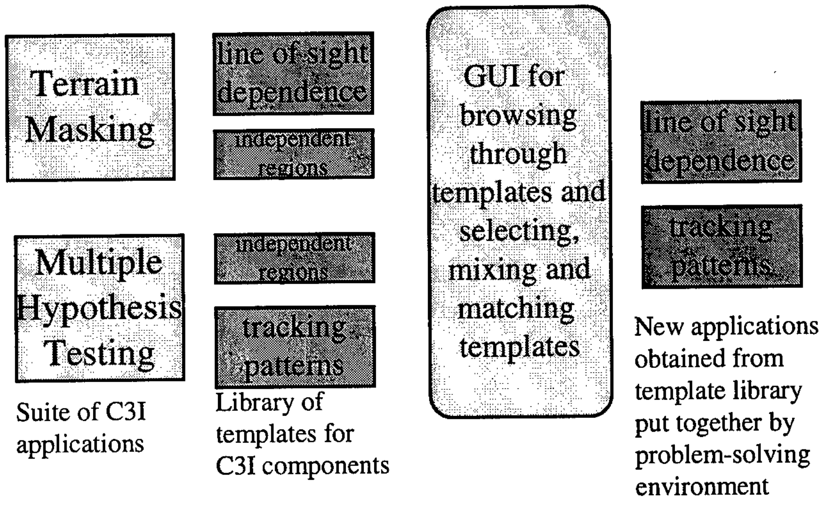
Recommendation

- Implement templates as data-parallel programs, allowing task-parallel composition of data-parallel components.
- Develop templates using an object-oriented framework, using inheritance or similar mechanism for tailoring a template to an application. Selection of language is open. eg. HPC++ or HPF.
- Map to NUMA architectures.
- Show use in different applications.
- Develop problem-solving environment (collection of tools) based on CVMs for building applications.

The Hardware Model

The hardware virtual machine is a network of symmetric multiprocessors. The terrain masking application can be run on a single symmetric multiprocessor, provided that it has adequate memory or a good IO structure. Even for a symmetric multiprocessor, data partitioning is important because of the role that cache plays in efficiency. The application can also be run on a regular network of uniprocessors such as the IBM SP2, or in a regular network of symmetric multiprocessors. The hardware-level virtual machine fits the terrain-masking application in a straightforward way.

Recommendation: The Problem-Solving Environment



Terrain Masking
Multiple Hypothesis Testing
Suite of C3I applications
line of sight dependence
independent regions
independent regions
tracking patterns
Library of templates for C3I components
GUI for browsing through templates and selecting, mixing and matching templates
line of sight dependence
tracking patterns
New applications obtained from template library put together by problem-solving environment

Integration of all levels of the virtual-machine hierarchy: The PSE

We recommend the development of a parallel C3I PSE. Next, we discuss the structure of such a PSE and its use for the terrain-masking application.

The PSE is designed for C3I applications and has a 3D+T+M problem space. The PSE is structured to support continuous evolution of C3I applications.

The PSE has a library of earlier implementations of a collection of C3I applications. This suite of applications will grow, and this suite describes the application space for which the PSE is appropriate. The PSE also has a library of archetypes for C3I components; examples of the archetypes are the two that arise in the terrain-masking application. A GUI that supports browsing through the archetype library helps in selecting appropriate archetypes for a component. The GUI also supports mapping on to different target architectures; this mapping is facilitated by the hardware virtual machine which provides a uniform framework for all target parallel machines. When new applications or archetypes are developed they are added to the libraries..

Summary

We recommend a hierarchy of virtual machines as opposed to a single virtual machine. The hierarchy allows mapping from the top-level virtual machine API to target parallel machines. It also supports user interaction in designing the mapping.

We recommend a specific virtual machine hierarchy. Our proposal for a concrete virtual machine hierarchy is based on converging technologies in C3I, distributed and parallel computation.

The recommended virtual machine hierarchy is defined at each level. Object-composition technologies are used to compose top-level objects. Parallel archetypes or patterns are used as the framework for application development. The parallel programming model is a task-parallel composition of data-parallel elements, and the hardware model is that of a network of symmetric multiprocessors. A PSE is used to integrate all levels of the virtual machine hierarchy.

Our experience with problems from the C3IPBS suite suggest that this approach is valuable. The next step is to develop the PSE that supports this approach.

APPENDIX 1: Architecture, application & user space

The goals of this report are to discuss the concept of virtual machines as applied to C^3I systems. The report discusses

- critical characteristics of C^3I systems,
- the virtual machine concept, goals for using virtual machines, and yardsticks for evaluating the success

of employing the virtual machine concept in C^3I ,

- trends in industry, and the potential impact of these trends on the use of virtual machines,

classes of users, applications and architectures that must be considered in designing virtual machines for C^3I

C^3I Systems

This section discusses critical aspects of C^3I systems. It identifies differences between militaryC^3I and other concurrent computing/communication systems.
The concept of virtual machines has been used in differentareas of computer science. We summarize the functions of C^3I and then list some of the important characteristics of military C^3I systems, with a view to gaining a better understanding of howthe virtual machine concept can be transferred over tomilitary C^3I systems.

Summary of C^3I Functions

A C^3I system supports:

1. **Information acquisition**: The acquisition of information from many distributed points, some of which may be mobile, and some of which may be underattack.
2. **Information transfer:** The transmittal of different kinds of information, and in different media (data, voice, still pictures, videos), to appropriate (possibly mobile) sites, in a battlefield situation.
3. **Information processing**: Correlation of information received with a knowledge base. Newly acquired information is fused with existing informationand the commander's perceptual model of the situation.
4. **Decision support:** Use of simulations, probabilistic analysis, mathematical programming and expert systems and other tools to support the decision-making process.
5. **Information display**: Information must be displayed in a way that helps commanders ask for additional information that they need, and thathelps them make decisions.
6. **Response to requests for additional information**: A commander may request additional information about an object.and C^3I systems must respond in a timely manner.

Though many commercial systems also support these activities, commercial systems can be simpler for several reasons. Military systems have to deal with powerful, unpredictable adversaries, with a variety of environments at all points of the globe, and short time frames. Commercial enterprises have a more predictable, more law-abiding environment, and longer time frames.

Characteristics of C^3I systems

The differences between the functions of military C^3I systems and commercial enterprises leads to some important differences in the characteristics of C^3I systems and commercial systems. These characteristics play a role in the design of virtual machines appropriate to these systems.

1. **Uncertainty:** C^3I systems must function in the ''fog and friction'' of war. The environment of C^3I systems is much less predictable than the environment for other concurrent systems. A design option in many commercial applications is to eliminate uncertainty, or at least, reduce it to the point where the uncertainty is not critical to decision making. This option is not available to military C^3I. A complete true picture of a battlefield cannot be obtained at all times. A design problem for C^3I is to cope with uncertainty rather than eliminate it.
2. **Timeliness** is also an issue in commercial applications, but it is less critical.
3. **Fault tolerance** C^3I must function even when an enemy is attempting to destroy it. So fault tolerance, and graceful degradation (some functionality even in the presence of faults) are critical.
   Mobility Military units and commanders can move rapidly. Channels of communication, computational power and storage capacity must move as well. Requirements for portability and ruggedness can limit options in the military domain. Though mobility is becoming increasingly important in commercial situations, the rapidity of movement in the military and the need for maintaining an ''infosphere'' around a mobile commander makes the problem more acute.
4. **Heterogeneity** Military C^3I systems use a variety of components --- sensors, communication channels, portable computers, actuators,... C^3I systems {\em have to deal with heterogeneous equipment}. By contrast, many commercial systems have the option of supporting only a small number of computational and communication platforms.
5. **Security** Military C^3I systems have to remain secure against intelligent adversaries with powerful equipment; this is a less critical issue in civilian systems.
6. **User interfaces** C^3I systems must present a huge amount of different kinds of information (weather at different points, terrain data, locations of enemy units,....) in a manner that helps commanders to (a) ask for additional information that they need and (b) make effective decisions in a timely manner. The variety of data and the wide range of types of data representation are more pronounced in military C^3I systems than in commercial systems.

7. **Knowledge Base** The information acquired by the system may not be completely accurate, and so each item of information must be correlated with other information and the commander's model of the situation. The knowledge base underlying the perception model is large and complex, and so the correlation of the model with new information can require significant amounts of processing. Though correlation with perception models is also carried out in commercial applications, situations are more predictable and information is more accurate.

8. **Decision Support** C^3I systems provide decision support to commanders to help them make timely decisions. This support can be of many forms including simulation, probabilistic analysis, expert systems and other Artificial Intelligence systems, and operations research tools. The complexity of the decisions, and the speed with which they have to be made, results in the need for powerful computational and data storage units for decision support.

Virtual Machines

This section presents the central ideas underlying the virtual machine concept, and relates these ideas to C^3I.
Information technology changed relatively slowly before 1980, but dramatic and rapid changes have occurred since then, and these changes are likely to continue. A challenge is to maximize the benefits of the government's investments in C^3I, even as information technology changes rapidly.

One approach is to design C^3I applications for a virtual machine that can be mapped to different target architectures; the investment in C^3I software is preserved to the extent that the virtual machine hides changes between architectures and that mappings from the virtual machine to target architectures are efficient.

Potential Benefits:
There are two main benefits to a virtual machine for C^3I.

- First, a virtual machine provides a specific domain of discourse. All the people who deal with C^3I, from experts on military doctrine to programmers can use the virtual machine model of C^3I as a common framework for discussion. To the extent that the virtual machine captures the most critical aspects of all C^3I systems, a debate about C^3I can use the virtual machine as a common foundation.
-
- Second, a virtual machine can help systems designers and programmers in stepwise refinement of their designs, by first focusing their designs for the virtual machine, and later refining their designs from the virtual machine to specific target machines. To the extent that the virtual machine captures the essence of C^3I and ignores details, this approach is cost-effective in dealing with designs for a large number of computational and communication platforms.

Costs:

- A virtual machine is an abstraction of reality. A cost with using a virtual machine is that perceptions of reality may be forced to fit the abstraction; we may attempt to force views of C^3I into the single mold of the accepted virtual machine. Even if a virtual machine is an adequate abstraction at this point in time, rapid changes in information technologies may result in today's virtual machine becoming obsolete; coercing our views of C^3I into an obsolete virtual machine can be very costly because it inhibits appropriate use of new technologies.

- Designing using a virtual machine requires two or more stages of stepwise refinement of designs: in the first stage, we design for the virtual machine, and in later stages this design is refined for the real target machines. This multistage design requires discipline, and can take more time than designing directly for the target machine.

The value of the virtual machine concept depends on several factors. The common theme running through all these factors is generality versus specificity.

- **How many virtual machines**? A single unified virtual machine that forms the basis of all C^3I design has the advantage of uniformity. A single virtual machine may not be adequate to handle the variety of tasks carried out in a C^3I system. If, however, a large number of virtual machines are required to adequately represent all components of a C^3I system, then the advantage of using virtual (as opposed to real) machines is reduced.

- **Level of abstraction**: A detailed virtual machine, at the level of a real machine and a real programming language, has the benefit of providing a single model all the way from initial design down to programming and testing. The difficulty is that the more detailed the virtual machine, the less likely it is that it can be used effectively to design for different platforms and technologies.

- **User Space**: Who are the primary users of virtual machines? The potential users range from researchers in military doctrine to programmers writing device drivers or to computational scientists doing weather modeling. We must determine for whom C^3I virtual machines should be designed. This is discussed in greater detail later.

- **Architecture Space**: The space of architectures that play a role in C^3I is huge. To help in the discussion we consider two broad categories: communication and computation.

  The range of options offered by commercial communications companies is exploding. These options, and many more, are appearing in military communications as well. The options include digital networks that support multimedia, and these networks are available at a variety of bandwidths. Satellite and cellular offerings are already becoming competitive. Commodity Personal Digital Assistants and laptop computers with wireless ports support mobility in the civilian world.

The range of architectures in computation is also large, and spans personal digital assistants to supercomputers. We propose a set of architectures that are most important from the point of view of C^3I.

- **Application Space:** A C^3I system is composed of many types of applications from AI programs for decision support, to battlefield simulations, to communication protocols, database manipulation, and visual representation of information. We propose a set of applications that are most important
from the point of view of C^3I.

Trends in Information Technologies

There are several trends in information technologies that will have an impact on C^3I systems and will indirectly impact the design of virtual machines for C^3I. Some of these trends are discussed next.

Commodity computational platforms and software The price differential between commodity processors and custom processors is likely to remain large due to the costs of fabrication and testing.

Price differences between commodity software (for operating systems, user-interface builders, word processors, databases...) and custom software is astronomical, and this price difference is having a major impact on the continuing trend towards client/server computing. The price/performance offered by commodity processors and software will influence the design of C^3I. The design of virtual machines for C^3I must pay attention to the client/server and commodity trends.

Explosion in bandwidth and communication options: The bandwidth offered by communication utilities to their customers is exploding. Some customers will be able to lease long lines at tens (and possibly hundreds) of gigabits from the phone companies and other providers in the next five years.

Companies with satellite-based communication are already offering mobile PC to host, and PC to PC, connections. Low bandwidth interactions are supported now by cellular technologies.

Continuing improvement in price/performance in computation: Computational power in millions of instructions per second per dollar, or millions of floating point instructions per second per dollar, or gigabytes of memory per dollar, will continue to decrease steadily. This decrease will make {\em sequential} computers extremely powerful. The increasing power of sequential computers will make sequential computers or shared-memory multiprocessors (with small numbers of processors) adequately powerful for some applications; highly scaleable systems with hundreds or thousands of processors will be needed only for applications that need huge amounts of power.

Civilian C^3I: Civilian C^3I is now a possibility because of the commoditization of processing hardware, communication hardware, and software for both computation and communication. Electric, gas and even telephone utilities, that get telemetered data from many remote sites, and manage complex networks as well as mobile maintenance crews, are examples of civilian organizations that have C^3I, though not at the level of the military. Recovery from emergencies (storms, earthquakes) for these utilities requires command and control in unpredictable situations. Civilian C^3I will be another factor driving competition, and this will eventually reduce prices for communication and computation systems.

CORBA, GDMO and other Object Interfaces An important trend in the computing and communications industries is {\em standardization around objects}. Interoperability between programs written in different languages and running on different types of platforms is achieved by treating a computational system as a collection of distributed objects with well-defined interfaces. Object interfaces are written in an IDL (Interface Definition Language), and the common IDL guarantees that objects can be composed. CORBA goes beyond object interface specifications by providing a common definition of an ORB (Object Request Broker) that allows objects to search for other objects with specified attributes, and to then link with these objects dynamically, with the interfaces being independent of program location.

Compilers from the IDL compile to programming language stubs in C, C++, Smalltalk and other languages; thus IDLs will allow systems to be composed from heterogeneous objects. Most vendors plan support for CORBA. Communications devices can interoperate using similar IDLs. This convergence towards a distributed object framework --- in some sense, a distributed object virtual machine --- can have a significant impact on military C^3I.

Standardization in Telecommunications: The increasing use of SDL (Specification and Description Language) is an example of the power of industry-wide standards set by CCITT (the institution responsible for setting standards for the telecommunication industry). SDL uses a virtual machine implicitly. As the boundaries between computation and communication becomes increasingly fuzzy, we will have to pay attention to virtual machines used in the telecommunications industry to design virtual machines for C^3I.

Fault tolerance Commercial offerings of fault-tolerant software, such as ISIS [Birman 87], is another indication that civilian organizations are paying increasing attention to high-availability service. The underlying communication layer is hidden at the CORBA level, but it may be necessary to consider the communication layer in the virtual machine.

Industry is also paying increasing attention to authentication and security as commerce on the Internet becomes more common.

Mobility: Mobile computing is becoming a reality in commerce. Advances in commercial mobile computing will impact military C^3I, and therefore will impact the design of virtual machines.

Graphical interfaces and display technologies: The increasing use of visual displays and interfaces in commercial applications will impact military C^3I which also uses visual interfaces.

Summary

In summary, the most critical trends in technology appear to be (a) the continuing improvement in price/performance and size in computation, communication and memory of commodity (sequential) processors, (b) industry standards for protocols (e.g. ATM) for interoperability (e.g. CORBA, OpenDoc, OLE), programming languages (e.g. C, C++,...) which together form de facto virtual machines and (c) increasing commercial attention to aspects of C^3I systems such as fault tolerance, security and authentication, real time, mobility and ultra-reliable designs, and the consequent improvement in price/performance for C^3I systems.

Indeed, it is not too much of an exaggeration to say that there is already a virtual machine that is widely used: networks of concurrent objects.

Architecture Space:

Existing parallel architectures that have been used for C^3I applications include the following classes of machines:

- Custom-designed processors: Typically used for embedded system applications, custom processor designs were needed to satisfy stringent response time, fault-tolerance or similar performance criteria.

- Symmetric multiprocessor (or SMP) workstations: These are generally limited to a maximum of about 20 processors, and machines with 4-8 processors are the most common. They are tightly coupled processors that share a global memory. They have relative small local caches that are synchronized using traditional snoopy mechanisms.

- Scaleable parallel computers: Traditionally referred to as MPPs for massively parallel processors, these architectures can provide low latency (currently of the order of 10-30 us) communications for a network of hundreds (and perhaps thousands) of processors. The two primary subtypes are distributed memory and shared memory architectures. Distributed memory MPPs do not provide hardware support for maintaining coherency of data replicated across local memories of multiple processors. Shared memory machines provide hardware support, at some level, to treat the entire memory as a universally addressable address space.

- Networks of workstations: Individual workstations may be PCs or SMPs connected by high bandwidth networks using ATM or FDDI technology. Although typical latencies for these networks are larger (by an order of magnitude) than for MPP systems, the cost per processor is considerably less.

Current View on architecture space for C^3I

Driven primarily by the difficulty of upgrading software for custom-designed processors, most future C^3I\ applications appear to be targeting commercial-off-the-shelf (COTS) hardware, where possible. The rapid standardization that is occurring in processor design, operating systems, and communication interfaces seems to indicate that for a large variety of C^3I tasks, the most common parallel platform will be a network of COTS workstations connected by low latency, high bandwidth networks. The individual nodes may be sequential or multiprocessor workstations, and the network technology is likely to be based on ATM. Scaleable, tightly coupled, parallel computers will continue to be required for C^3I tasks like generation of command and air-battle planning, incorporating real-time data collection into mission prioritization, and data mining.

Mobility is also expected to play an increasing role as wireless communication software on laptops and other mobile computing elements becomes standardized. As mobility of computing elements adds significant complexity to the design problem, we expect that CVMs for a mobile computing environment will be sufficiently different from those for a stationary environment. We expect that the work covered by this project will have a significant impact on CVMs for mobile computing environments, but do not propose to directly address those concerns in this work.

Application Space

C^3I applications have spanned the entire gamut of parallel processing from scientific computations and manipulation and visualization of large databases, to battle simulations and decision support systems. These applications can be classified into the following primary categories:

- Sensor data acquisition and processing: Traditionally, design of sensor elements (infra-red sensors, transducers, ...), their data acquisition and processing, and external effectors (weapons, ...) have been important components of C^3I systems design. Thus projects that include numerical techniques for transformation of sensory data, multi-sensor correlation with existing track databases, fast Fourier transforms, and other numerical techniques to process sensory data have been viewed as part of the C^3I application domain. Track assignment in Radar systems, infrared sensor data collection, Fast Fourier transforms and inverse transforms are some other examples of applications and activities in this category.

- Image understanding: Feature extraction, pattern recognition, edge detection and related applications are examples.

- Scientific computations: A large class of scientific computations fall within the purview of C^3I applications. Commonly used kernels include matrix operations, factorizations, and linear equation solvers for dense and sparse matrices. Libraries for partial differential equation solvers using iterative, spectral, and multi-grid methods are also becoming common.

- Combinatorial optimization: Many C^3I\ tasks involve path planning, route optimizations, and related computations for heuristic optimizations. Some of these computations may be on-line in nature, representing changes in the underlying connectivity (existing paths may be destroyed and new ones created) or in the resource demands.

- Hard real-time systems: hard real-time systems with guaranteed response times that include many process control and human-computer interaction systems. SAR processing and threat analysis are common examples in this category.

- Performance-constrained systems: These include a wide variety of systems that must satisfy a host of additional constraints that may include size constrains, power dissipation requirements, radiation and temperature hardening, fault tolerance, and maximum response time or minimum throughput guarantees.

- Simulations: Battle and scenario simulations have perhaps been one of the most important C^3I\ applications. These require a wide-range of models including continuous and discrete-event models, some of which may have real-time constraints, require human interactions and provide innovative data displays.

- Decision support: off-line data processing and integration to formulate plans and on-line prediction of impact of plan modifications is an important component of many C^3I\ applications and includes many AI technologies including rule-based systems, heuristic tree searches, and probabilistic reasoning. Communication protocols: Reliable and secure protocols for communication among diverse computational units linked by heterogeneous communication media are central to the operation of many C^3I applications. Design and deployment of protocols for multimedia communications in a dynamic environment is of particular importance.

- Data query and retrieval: Databases accessed by a C^3I\ application may range from weapons and personnel inventories to massive geographical databases for terrain recognition.

- Visualization: innovative techniques for displaying relevant data have resulted in many new technologies including virtual reality and the proposed infosphere.

Current View on application space for C^3I

Rather than address all of the multi-faceted aspects of C^3I applications enumerated above, the application focus of this project is on designing a CVM for C^3I-related computations that emphasize coordination. Thus, it is not our intention to provide the tools needed for specification and design of scientific computations, image processing, and other computational subtasks directly in the CVM. Such activities typically comprise a self-contained, tight loop of functionality which does not require interaction or feedback from the rest of the system, once they are initiated.

From the perspective of a CVM-based design methodology, the preceding class of activities can be abstracted in terms of its gross resource requirements measured in terms of CPU, IO, and communication bandwidth requirements. Our approach in the CVM design will be to focus on the synchronization and coordination aspect of the C^3I application, relegating the specific computational aspects to domain specific CVMs, if warranted.

Focus of the CVM

Driven primarily by the difficulty of upgrading software for custom-designed processors, most future C^3I applications appear to be targeting commercial-off-the-shelf (COTS) hardware, where possible. The rapid standardization that is occurring in processor design, operating systems, and communication interfaces seems to indicate that for a large variety of C^3I tasks, the most common parallel platform will be a network of COTS workstations connected by low latency, high bandwidth networks. The individual nodes may be sequential or multiprocessor workstations, and the network technology is likely to be based on ATM. Scaleable, tightly coupled, parallel computers will continue to be required for C^3I tasks like generation of command and air-battle planning, incorporating real-time data collection into mission prioritization, and data mining. The primary focus of this project is on virtual machines for parallel systems that form nodes of larger distributed systems; a secondary focus is the entire distributed system. Although mobility is not a focus of this research, we expect that the central ideas proposed by this research should scale from PDAs and laptops to supercomputers.

From the user perspective, the basic ideas of the CVM should be usable at different levels, from conceptual system design down to actual coding. However, the focus of this research is on systems designers and programmers. Military strategists and hardware design is not a concern.

From the application perspective, although C^3I applications cover much of the application space spanned by parallel programs today, the focus of this project is on designing a CVM for C^3I-related computations that emphasize coordination. This includes support for CVM including expert systems, databases, and decision support for C^3I. However, it is not our intention to provide the tools needed for specification and design of scientific computations, image processing, and other computational subtasks directly in the CVM.

In summary, current trends in technology indicate that it is perhaps not much of an exaggeration to say that there is already a virtual machine that is gaining acceptance in the context of parallel computations: networks of concurrent objects. The next phase of the project will undertake a survey of the state of the art for virtual machines for C^3I applications.

APPENDIX 2: APPLICATION DEVELOPMENT AND PORTABILITY USING A CONCURRENT VIRTUAL MACHINE

OVERVIEW: A COST-BENEFIT ANALYSIS

In this manuscript, we discuss the concept of using a*concurrent virtual machine* to port high-performance computer applications from one platform to another. A concurrent virtual machine is an abstraction for handling concurrency, for ease of movement between platforms; rather than having the notion of a single concurrent virtual machine, we present an arsenal of many different concurrent virtual machine models. Each of these models have some features they share with other concurrent virtual machine models, and some unique features as well. Using this information, an application developer can make an informed decision as to whether to use a particular concurrent virtual machine in program porting, and an intelligent selection of which concurrent virtual machine(s), if any, is most appropriate.

When we refer to "platforms" or "machines", we consider the migration of applications across:

- Architectures (i.e., computers, processors, and networks)
- Operating systems
- Programming languages
- Higher-level notations
- Interoperable objects
- Application-level component environments and suites

Sometimes the desired application port requires the migration of applications across several platforms simultaneously, and sometimes it requires the migration of applications onto new target platforms. This document assembles in a single volume the issues involved when considering cross-platform porting of applications.

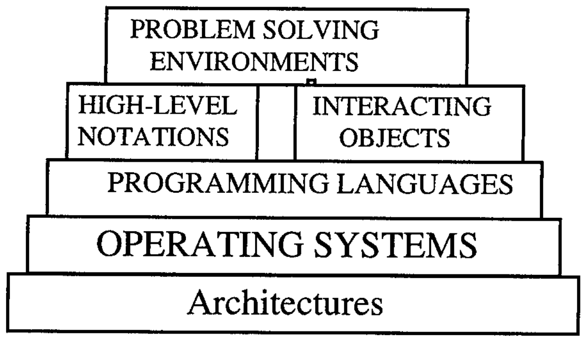


Figure 1: CVM Hierarchy

It should be noted that the platform types mentioned above form the layers of a hierarchy of a concurrent virtual machine, as illustrated in Figure 1. In addition, each of these layers can be considered a concurrent virtual machine in and of itself, as we discuss in the respective chapters for each of these layers.

As we discuss throughout this document, each of these layers of the concurrent virtual machine hierarchy should be considered when an application is to be ported across platforms. The main concern of our discussion is the tradeoff between the ultimate costs and benefits of the proposed application port:

> **The *cost-benefit analysis* should be performed at all of the levels of the platform hierarchy to determine if porting the application is a worthwhile use of resources.**

Cost-benefit analysis is an efficient method of deciding whether an application port will be worth the time and money required. The costs and benefits that can be analyzed independent of the porting platform(s) are detailed in this section, and specific platform layer costs and benefits are described at the beginning of each respective section.

A concurrent virtual machine provides an intuitive intermediate level of abstraction for computer applications, which can greatly reduce the effort a programmer spends porting an application to a new parallel architecture. This middle abstraction level (as illustrated in Figure 2) allows for ease of program movement from the source machine to different target architectures. However, we emphasize that a programmer should carefully consider the value of designing with a virtual machine before determining the method's suitability for a given project. In this section, we review the costs and benefits of the virtual machine approach to design. The cost-benefit analysis for using virtual machines resembles the analysis a programmer would undertake in considering different high-level languages.

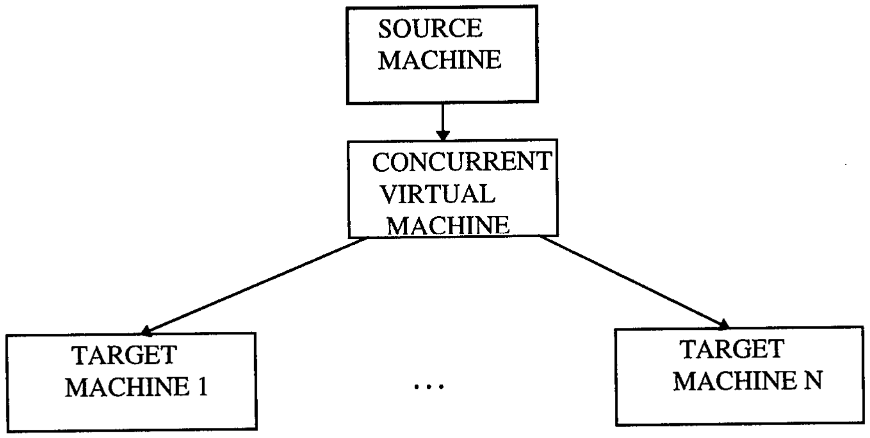


Figure 2: Porting an application using a concurrent virtual machine

Benefits of using Virtual Machines

Preservation of the initial design investment.

A virtual machine's key benefit is its conservation of the application's design, regardless of the transition to a different architecture: The design and code developed for the virtual machine remains unchanged when the target machine is changed. This benefit is especially significant when the application must be ported to several distinct target architectures; throughout the series of program ports, the virtual machine remains unchanged.

Provision of a high-level abstraction medium.

In addition, a virtual machine provides a higher level of abstraction than an actual machine. This observation allows a compiler from a virtual machine to the real machine to address certain architecture-specific details, freeing the programmer from worrying about some of those lower-level particulars. Hence, programs for a virtual machine are simpler than their target machine counterparts, and require less development effort.

Reduction of program maintenance time.

This higher level of abstraction also can reduce program maintenance effort, since many application specifications are altered frequently over time. Without a virtual machine, as new functionality are added, the incorporation of additional specifications might require more work than the initial development did. High-level virtual machines can reduce the time needed to implement such modifications.

Costs of using Virtual Machines

Consideration of the tradeoffs.

A virtual machine may not exactly "fit" a real machine, since the virtual machine is an abstraction that might not be appropriate for a given machine. An abstract design might become more complex if it accounts for multiple target machines with portability kept in mind; in such a situation, the programmer must contribute more effort than that required simply designing for a single actual machine. However, this cost is not unique to the virtual machine idea: Any application port between dissimilar architectures will require substantial effort in locating and handling program issues.

Complication of efficient porting.

Porting programs developed for a virtual machine from one real machine to another real machine can be difficult, because the source code produced for the target machine might not be sufficiently optimized for that architecture. This cost is measured by comparing the efficiency of the code produced by the virtual machine's automatic mapping, to code written specifically for the target machine. Again, this cost is not unique to the virtual machine idea: Any application port between dissimilar architectures will require substantial effort in locating and handling program issues. However, the virtual machine concept introduces the additional cost of developing the software to port code for a virtual machine onto a real machine.

A Cost-Benefit Model: Yardsticks for using Virtual Machines

When considering a methodology for a given application port, the costs and benefits of designing using virtual machines can be compared in a simple way, using the following model. These tradeoffs must be carefully weighed to determine the suitability of using a virtual machine for a given project.

Use of virtual machines: Benefits.

- Development effort reduced by salvaging aspects of the initial application design.
- Design effort reduced by the higher-level of abstraction afforded in programming a virtual machine, as compared to developing programs for a specific real machine.
- Maintenance effort reduced by using the same virtual machine designs and programs on a variety of different real machines.

Use of virtual machines: Costs.

- Additional effort required (if any) for developing programs for a virtual machine, as compared to developing programs for a specific real machine.
- Programming effort required to develop a mechanism for compiling and mapping an application from a virtual machine to a real machine.
- Effort required to achieve desired level of program efficiency ported from a virtual machine to a real machine.

Current Approaches to Virtual Machines

Porting an application from one sequential machine to another is a much simpler task than porting from one architecture to another, for two reasons. First, all sequential machines have a von Neumann architecture, and a palette of supported common high-level languages (e.g., Ada and C [HOPL 93]). To port an application written in such a language, a programmer simply compiles the high-level source code into chip-specific machine code. Second, a sequential machine has only a single computing node, so a sequential program does not need to address such issues as communication between nodes, scheduling of tasks onto nodes, and distributing memory accesses.

Even so, it is instructive to compare existing approaches for porting applications written in an *unsupported* sequential language from one machine to another, with the proposed approaches for using virtual machines for porting applications from one architecture to another. A few of these existing approaches are

- *Emulation.* Porting an application written in language that is no longer supported, from one machine to another, is difficult; it is also a widespread problem in government and industry. One approach is to develop an emulator that interprets programs in the unsupported language in terms of a widely-supported language. Emulation can be slow because interpretation can be slow; nevertheless, emulation on new hardware is often faster than execution on old hardware.

- *Compilation.* A related approach is that of using a source-to-source compiler from the unsupported language to a supported language. In both cases, the virtual machine is the programming language, and porting is achieved by emulating or compiling instructions in one virtual machine to the other machine.

- *Reengineering.* The process of reengineering requires that a programmer use tools to develop an understanding of the original program, and to structure the program into components. The programmer then rewrites the components in the target language and tests them against the original code. In this case, the problem is the comprehension of the original application and virtual machine, and redesigning the application for a new virtual machine.

These three approaches illustrate the essential idea of virtual machines: to have a layer of specification common to a large number of applications, so porting all the applications can be achieved by porting the common layer. This is already being done in DOD and elsewhere, by using standardized programming languages like Ada and C. These standardized languages serve as virtual machines for sequential and distributed machines. The task at hand is to achieve the same level of portability across machines with very different architectures.

Tradeoffs in Virtual Machine Design

Several issues deserve consideration in the design of virtual machines, including

1. *Level of concreteness.* A programming language with an associated performance model is an example of a concrete virtual machine. An application that uses this virtual machine (the programming language) has a well-defined semantic; the application can be executed, and its efficiency can be estimated. Other concrete virtual machines include Common Object Request Broker Architecture (CORBA) [OMG 95] implementations such as Distributed System Object Manager (DSOM) [SOM 94] from IBM; for DSOM, the virtual machine is not a programming language but a system with well-defined semantics and syntax for composing binary objects. Less concrete virtual machines are descriptions such as the CORBA specification itself. Here the virtual machine is well defined, but applications on the virtual machine cannot be executed. Even less concrete virtual machines are models such as the Parallel Random Access Machine (PRAM) [Fortune 78] model that deals only with performance of a limited number of algorithm types.

The virtual machine concept mandates that developers program for a common layer; however, a wide range of such layers are feasible, and the most suitable layer must be chosen. Concrete virtual machines are advantageous for straightforward porting of an application developed for that virtual machine to another concrete virtual machine. Less concrete virtual machines are advantageous because they can fit a variety of more concrete machines (e.g., the CORBA specifications fit both DSOM and Distributed Objects Everywhere (DOE) by SunSoft).

2. *Level of abstraction.* A specification notation, with an associated performance model, is a virtual machine at a high level of abstraction, whereas an assembly language has a relatively lower level of abstraction. However, both notations (as well as all the notations with abstractions in between) are, in effect, virtual machines. Likewise, in the performance area, we can have high-level models that treat performance merely in terms of Big-Oh notation (e.g., O(n) counts of the numbers of operations [Knuth 76]), and we can have low-level detailed models that deal with performance of such operations as message-passing primitives and operating system calls.

3. *Quantity of virtual machines.* The virtual machine concept does not restrict designers to employ only one virtual machine. Designers can use a tree of virtual machines, with high level, less concrete virtual machines early in the design, and more concrete, lower-level virtual machines later in the design. This way, it will be possible to reuse effort to design for a virtual machine at some level of the tree, even if designs for lower-level virtual machines cannot be ported to a new architecture. The disadvantage of this approach is the overhead associated with dealing with multiple layers, coupled with the difficulty of obtaining a consensus about the definitions of these layers.

4. *Separating design concerns by using virtual machines.* A contribution to the complexity of applications is the entangling of issues such as performance and real-time issues, fault-tolerance, and functional correctness (i.e., independent of performance and fault-tolerance). Virtual machines can help in disentangling these issues and thus simplifying designs. A virtual machine can deal with the functional specification issue, another virtual machine can deal with the performance and real-time issues, and yet another virtual machine can deal with redundancy and fault-tolerance issues. The idea of layers in protocols has been very successful; the idea of layers of virtual machines to deal with different concerns can be equally successful.

Next, we discuss different layers of computing and communication, from the lowest level of abstraction on up, and we discuss the relevance of these layers to virtual machines. These layers are hardware architectures, operating systems, programming languages, interoperable layers for distributed systems, specification layers, and application layers.

ARCHITECTURE ISSUES FOR VIRTUAL MACHINES

This section contains a brief overview of current trends in parallel architectures and networks. We discuss computer architectures, and then consider communication architectures. For each architecture, we discuss its present status and trends, and then discuss its relevance to virtual machines. We note that excellent overviews of architectures are contained in [Patterson 90, Leighton 92, Amalsi 94, Morse 94] and an excellent source about networks is [Tanenbaum 89].

The Cost-Benefit Model: Yardsticks for porting across Architectures

When considering an architecture port, the costs and benefits of designing using virtual machines can be compared using the following model. These tradeoffs must be carefully weighed to determine the suitability of using a virtual machine for this layer of a given project. The following questions are relevant for application developers when considering different architectures:

- Is the machine architecture sequential or parallel?
- If parallel, is there a single instruction stream or are there multiple instruction streams?
- If parallel, is the memory shared, distributed, or a hierarchical hybrid?
- If parallel, is the programming model shared-memory or not?
- Is the system geographically distributed across a wide-area network, loosely coupled across a local-area network, or tightly coupled in a single real machine?

Computer Architectures: An Overview

The first supercomputers provided parallelism that was essentially transparent to the programmers and relied heavily on compilers (although by following certain guidelines a programmer could improve the quality of code that was generated by the compiler). Important ideas included the use of *vector instructions* that used pipelining to execute a given operation on vector rather than scalar operands, instruction pipelining that overlapped the fetch of the next instruction with decoding of the current instruction, and the use of multiple functional units. In effect, the virtual machines that were used for sequential architectures did not have to change significantly to accommodate these advancements in processor designs.

Although many of the preceding design innovations are an important component of contemporary parallel architectures, the primary emphasis is on the methods used to interconnect multiple processors that are directly visible to the programmer. These architectures fall into two broad classes: Single-Instruction, Multiple Data (SIMD) and Multiple-Instruction, Multiple Data (MIMD). SIMD architectures execute a single stream of instructions, in parallel, on multiple data items. As the instructions for the program can be broadcast from a single node, it is not necessary to store the program individually on each processor allowing each processor to be relatively inexpensive.

In MIMD architectures, each processor can store its own program and data, where its program may be distinct from the program stored in other processors. Naturally each of the processors of an MIMD architecture is more complex than the processor in an SIMD machine. Each of these classes of parallel architectures have spawned their own models of parallel programs (and corresponding virtual machines). Another relatively recent development in the domain of parallel architectures is the advances in networking technology that have made it feasible to use a network of workstations or PCs as a supercomputer. The individual machines in the network may be distributed across a building or even across a continent and linked using a local, metropolitan, or wide area network as appropriate.

Sequential Computers

Sequential computing price/performance continues to improve, and there appears to be no technological show-stopper for the next 5 to 10 years. Likewise, the number of gates per chip, continues to increase. RISC processors with multiple instructions executed concurrently, with effective uses of caches, are becoming common.

From the point of view of virtual machines, the technology path for sequential computers for three decades has used a single common virtual machine (i.e., the Von Neumann computer), and this is one of its greatest strengths. Programming languages, such as C, COBOL and FORTRAN, developed in the 1960s and 1970s, continue to serve as the predominant "virtual machines" for this century, and will likely continue to do so for the next 10 years. Sequential computers and their associated languages are a success story for the virtual machine concept.

Timesharing systems lead to virtual machines which dealt with multiple processes and multiple threads within a process. Each process has a separate address space; threads within a process share the same address space. The operating system provides methods to: create threads and processes, send messages between processes and synchronize between threads, schedule processes and threads, and terminate them. Thus, even with sequential hardware, virtual machine layers based on processes and threads were developed, with this layer provided by the operating system.

Parallel SIMD Architectures

Early parallel SIMD architectures used the same virtual machine as sequential architectures and relied on compiler generated parallelizations. More recently, architectures like that of the Connection Machine CM-2 [Leiserson 90] connected many thousands of simple processors which led to the development of the data parallel programming paradigm. There are relatively few applications, primarily in the dense matrix computation domain, that appear to have the characteristics necessary to effectively use a large number of processors in the SIMD mode. From the C^3I perspective, these architectures appear to have limited applicability as they are unsuitable for the execution of irregular command and control applications.

Parallel MIMD Architectures

Most contemporary parallel architectures (and those likely to be in wide use in the next decade) fit the MIMD model. These architectures can be further subdivided into two broad classes: shared address-space (or shared memory) machines and shared-nothing (more commonly called message passing or distributed memory) machines. In the former category, each processor can directly address any memory location, whereas in the latter each processor has its own local memory and cannot directly access the memory attached to another processor.

The primary differences in many of the preceding machines lie in the design of their interconnection network that is used to connect the multiple processors. Available interconnection networks are broadly classified as*static* or *dynamic*. In a static network, processors are directly connected to each other using a fixed topology like a ring or mesh. The communication primitives supported in hardware directly reflect the static connectivity. Primitives to support communication between arbitrary pairs are typically provided by a software layer built using the available hardware primitives. Static networks are typically used to construct message passing architectures. In a dynamic network, communication links between processors and memory modules are connected dynamically using switching elements. In other words, it is possible to provide hardware support for communication among an arbitrary processor, memory pair. These networks, also called indirect networks, are typically used to construct shared address-space machines.

We consider each of these architectures below together with their possible impact on virtual machines for C^3I applications.

Parallel Shared Memory Architectures

The earliest type of shared memory machines were the Uniform Memory Access (UMA) machines, where a processor did not have local memory (other than the standard on chip cache memory), and all memory accesses had essentially the same cost. However these machines were not scaleable and typically had no more than a few tens of processors. The most commonly available UMA architectures today are the so called Symmetric Multiprocessor (or SMP) workstations and personal computers (e.g., SGI Challenge series, SUN Sparcsation 1000, and similar products from HP). The interconnection architecture used in these machines is typically a bus which provides a low cost uniform interface but restricts the maximum number of processors that can be supported to a relatively small number. To increase the number of processors that can be connected in the UMA model, some architectures have used multistage interconnection networks. This network uses a number of stages to connect the processors to the memory modules, where each stage has the same number of input and output links (or ports). Multistage networks differ in the number of unique paths that they provide between a given processor and memory pair and also the number of stages that must be traversed for each access. The omega network is perhaps the best known multistage interconnection network. This network provides exactly one path between any processor and memory module, where every path traverses *log p* stages for an architecture with *p* processors (and memory modules).

To further increase the number of processors that could be supported in a shared address-space architecture, recent efforts have suggested the use of Non-Uniform Memory Access (NUMA) machines. NUMA architectures are similar to message passing architectures in that each processor may have local memory in addition to the shared memory, such that the access time for a remote memory may be longer than that for local memory. However unlike message passing architectures, these machines provide hardware support for a processor to access any memory module in the machine. Perhaps the most well-known NUMA architecture is the Stanford DASH [Lenoski 92], which uses a directory based scheme to manage shared data.

Virtual Machines

From the point of view of virtual machines, multiprocessors require a virtual machine layer that deals with threads. At the very least, this layer has to deal with mechanisms for creating threads, synchronizing and scheduling threads, and terminating threads. Many (if not most) operating systems developed in 1995 and beyond will support threads. This virtual machine layer is now beginning to be supported in programming languages with language mechanisms for creating, synchronizing and terminating threads.

The "threads virtual machine" predates widely-available multiprocessors. However, the increasing availability of multiprocessor PCs and workstations has given an impetus to making this virtual machine common at the operating system level, and is making this virtual machine apparent at the programming language. Threads-based application frameworks may well become common in the next five years.

Parallel Distributed Memory Architectures

Parallel machines in this category are often referred to as multicomputers [Athas 88, Seitz 90], which is the term we will use henceforth. The earliest multicomputer was the Caltech Cosmic Cube [Seitz 85] which connected a number of processors using a static network with the hypercube topology. Other topologies that have been used in more recent designs include meshes (e.g., Intel Paragon [Intel 91]) and trees (e.g., Thinking Machines CM-5 [Thinking 91]).

As the hardware in a static network only supports communication on point-to-point links of the static network, additional message routing techniques must be provided to support efficient communication among arbitrary source and destination pairs. The most common techniques to route messages from one node to another over multiple links are the*store-*

and -forward routing and the *cut-through* routing. In the former, the entire message advances one link at a time; an intermediate node waits until it has received the entire message from its predecessor before forwarding it along the next link. In contrast, with cut-through routing, an intermediate node starts forwarding an incoming message as soon as it arrives.

The most well known type of cut-through routing is called*wormhole routing*, which pipelines the incoming message on the outbound communication link as the first bits (or *flits*, for flow control digits) arrive at an intermediate node. The efficiency of message communication on static networks is determined by the hardware and software overheads of message transmission. The hardware factors include link latency or the hardware transmission time for a single byte, average number of links on each path and the network bandwidth or the total number of bytes that can simultaneously be pumped through the interconnection network. Software overheads include the time to transfer the message from the user space to the communication buffer at the sending end, route computation, message preparation (adding control information to the header etc.), and the overheads at the destination processor in transferring the message from the incoming communication buffer to the user space. The hardware overheads depend primarily on the design of the network topology whereas the software overheads, which clearly dominate the message transmission costs depend on a number of factors including protocol design, the network-operating system interface, and the application-operating system interface.

In the past, the interfaces had to be custom designed to minimize the communication overheads . However recent advances in the design of communication switches make it possible to connect standalone workstations using the same switch technology used for distributed memory computers. This has led to the emergence of workstation clusters, also called *Networks of Workstations* (NOW) architectures [Patterson 94] as possible platforms for parallel computing.

Virtual Machines

From the point of view of virtual machines, multicomputers require a virtual machine that supports processes; the virtual machine must have mechanisms for creating processes, communicating between processes and terminating processes. Many virtual machines have been used for multicomputers. The simplest is a Single Program Multiple Data (SPMD) machine, which does not support dynamic creation and termination of processes; the programmer works with a fixed collection of processes (usually, but not necessarily, one process per processor), and a message-passing or Remote Procedure Call (RPC) [Birrell 84] layer. The more complex virtual machines support dynamic creation and destruction of processes and communication channels between processes.

Programming Models

The shared-memory and distributed memory dichotomy among MIMD architectures has led to two distinct programming models: the shared-everything and the shared-nothing models. Each model includes the notion of processes but differs in how the processes interact with each other. In the former, the programmer views the program data space as a single globally addressable memory where there is no syntactic distinction between access to local or remote data. In contrast, in the shared-nothing model, the programmer must first decompose the program data space into *p* disjoint partitions, each of which represents data that is local to the corresponding processor, and the program must make a syntactic distinction between accessing local data and remote data (which is typically done using messages).

In principle, either of the preceding programming models may be implemented on both classes of MIMD architectures considered earlier. Clearly the shared-nothing model may be easily implemented on both architectures, where the shared memory implementation uses a shared data space to implement message passing. Although the shared-everything programming model is generally easier to implement on shared memory architectures, recent research in the areas of global shared memory implementations on distributed architectures have shown significant potential, particularly when used in conjunction with programmer annotations to aid the initial decomposition and subsequent migration of the shared data (e.g., the Munin programming system from Rice University [Carter 91]).

Hierarchical-Memory Machines

A hierarchical memory machine is, in its simplest form, a collection of multiprocessor computers connected by communication network. One reason for this architecture is that multiprocessor microprocessor chips are becoming available, and connecting collections of microprocessor chips on a communications backplane gives a hierarchical memory machine. Multiprocessor architectures do not scale to thousands of processors, whereas multicomputer architectures can; hierarchical memory machines provide scalability with some of the advantages of shared memory.

The virtual machine that exploits this hardware supports both processes and threads. Of course, it is possible to use a virtual machine that only supports processes and does not support threads, and that does not take advantage of the shared address spaces provided by the hardware. Likewise, it is possible to use a threads virtual machine that maps on top of the communication layer.

There are several ways of designing virtual machines that support both threads and processes. We discuss these later.

Geographically Distributed Systems

Geographically distributed systems such as battlefield (or worldwide) C4I systems are critically important to DOD. The virtual machines for these systems must deal with important features including:

- fault-tolerance: the system must operate in a hostile environment and elements of the system may be destroyed.

- timeliness: information must get to its destination in a timely manner.
- mobility: the network is dynamic.

Virtual machines for these systems are complex because they must deal with a variety of features. One approach to simplifying the design is to use virtual machines at different layers to deal separately with concerns of functional correctness, timeliness and performance, fault-tolerance, and mobility.

Networks

The rise of global networking has greatly facilitated the use of geographically distributed computers as an integrated computational unit. Although the first generation of computer networks were used primarily for transmission of audio traffic, contemporary networks are being used for multimedia transmissions that include voice, data, and video. The two most significant trends in modern networking are the move towards*gigabit networks* using the Asynchronous Transfer Mode (ATM) [Tanenbaum 95] and the growing popularity of *wireless* networks that may be integrated with wireline networks to support multimedia communications

As illustrated in Figure 3, the networking community adopted a*layered architecture* in the early years of networking. Perhaps the most well known of theses architectures is the seven layer OSI model [Tanenbaum 89]. The lowermost four layers in this model that are primarily responsible for transferring data from one node to the other are:

- *Physical* layer: The lowest level layer which represents the physical medium that is used to transmit the data.
- *Data link* layer: The next layer that is responsible for dividing the input data into frames and ensuring reliable delivery of frames.
- *Network* layer: This layer provides the media access control and is responsible for computing the route, congestion control etc. This is also the internetworking layer which is used in linking different networks together.
- *Transport* layer: This layer provides a communication stream between two applications that execute on different host computers.

The remaining three layers*session, presentation*, and *application* provide additional services like data format conversion and high level abstractions like RPCs.

The layered approach is rarely used for protocol implementations because it is often inefficient and involves duplication of services like acknowledgments that are common across many layers. However, the idea of layering has been invaluable for understanding the role played by various networking protocols and in defining their interfaces. The layered approach makes it relatively easy to define virtual machines for networks as many of the lower level issues can be abstracted by the corresponding interface.

APPLICATION
PRESENTATION
SESSION
TRANSPORT
NETWORK
DATA LINK
PHYSICAL

Figure 3: Network Layers

Summary

Different virtual machines are required to exploit features of different architectures. Changes in architecture are helping to create new kinds of virtual machines, and these virtual machines are being propagated, as we shall see, through the operating system and programming language layers to the application layer.

OPERATING SYSTEM ISSUES FOR VIRTUAL MACHINES

An operating system has four major components: process management, input/output control, memory management, and file system handling. The operating system directs all of a computer's resources, and furnishes a foundation on which application programs can execute. In this section, we demonstrate the relationship between virtual machines for operating systems and architectures, and later we illustrate the commonality of virtual machine concepts across programming languages, operating systems, and architectures.

The Cost-Benefit Model: Yardsticks for porting across Operating Systems

When considering an operating system port, the costs and benefits of designing using virtual machines can be compared using the following model. These tradeoffs must be carefully weighed to determine the suitability of using a virtual machine for this layer of a given project. The following questions are relevant for application developers when considering different operating systems:

- What are the mechanisms for creating, scheduling, running, and stopping processes?
- How do processes interact with each other and the environment?
- What process communication mechanisms are indigenous to the operating system?
- What are the mechanisms for creating, scheduling, running, and stopping threads of control within processes?
- How do threads interact with each other and the environment?
- What thread locking mechanisms are provided?
- How is resource contention handled?
- What is the architecture on which the operating system runs?
- Does the operating system manage a real-time system?

Operating Systems Concepts: Processes, Communications, and Threads

The virtual machine supported by operating systems for sequential computers is a process-based virtual machine. However, special care must be taken when allowing multiple threads of control within processes.

Processes

The virtual machines for operating systems such as UNIX are process-based [Tanenbaum 87]. Each process has a single address space and a locus of control; different processes do not interfere with one another. Processes can exist in one of several states: executing, blocking, ready, or terminated; in addition, processes are afforded a rich functionality that allows them to create child processes and interact with each other and the environment.

The operating system makes provisions for creating, scheduling, running, and stopping processes.

Process Communication Mechanisms

Processes communicate through the operating system by employing some mechanism such as message sending and receiving over a network using sockets [Tanenbaum 89]. Sockets are end points to which connections can be attached from the user side. When a socket is created, a data structure is maintained for it within the operating system, including the address format (such as Domain Name System), socket type (connection-oriented or connectionless), and protocol (e.g., TCP/IP). A created socket can buffer incoming connection requests. A socket can be named, and this bound name allows remote processes to issue requests to the socket. Other UNIX system calls allow processes to select, connect, send messages, receive messages, and perform shutdowns between sockets.

In addition, UNIX pipes can be used for interprocess communication by providing an intermediate file-like interface. Pipes differ from ordinary files in that input and output can block; as a consequence, the operating system manages the suspension and restarting of pipe commands as warranted. Services such as concatenation and redirection of bit streams can be enacted easily using pipes [Peek 93].

Multiple Threads of Control

In many distributed systems, it is possible to have multiple threads of control within a process. To allow an operating systems to manage these controls, threads libraries were introduced and later standardized [POSIX 90]. A thread is an independent locus of control within an address space. In most traditional operating systems, each process has an address space and a single thread of control; the introduction of multiple threads of control (also called lightweight processes) within a single process provides new functionality (and pitfalls) for the programmer.

In many respects, threads are like mini-processes: Each thread executes sequentially and maintains a program counter and stack. The operating system schedules threads to share a computer's CPU much like processes do; on a multiprocessor architecture, separate threads can run in parallel. In addition, if a thread is blocking, another thread in the same process can execute.

However, different threads in a process share exactly the same address space (and global variables). Since every thread can access every virtual address, one thread can read and write every other thread's variables and stack. Thread-safe communication mechanisms allow interprocess and intraprocess thread cooperation. An excellent discussion of threads and their operating system implementation is provided in Chapter 4 of [Tanenbaum 95].

Thread libraries were not initially integral parts of the operating system of sequential computers. A threads package is a set of library routines available to the user for programming with threads. The package offers primitives for static and dynamic thread management, thread creation and destruction, mutual exclusion (mutex), and condition variables. Mutex provides a short-term locking entry guard for allowing threads to manipulate shared data safely in critical regions. Condition variables are similar to mutexes, differing in functionality: They are used for long-term waiting until a resource becomes available.

Multiple threads of control are particularly suited to multiprocessor systems, where different process threads can execute on different processors in the architecture (as opposed to context switching in and out, as on a uniprocessor machine).

Operating Systems for Multiprocessors

Operating systems that can exploit multiprocessors include Microsoft's Windows NT, IBM's OS/2 and AIX, and versions of UNIX such as Solaris and Mach. The virtual machine supported by these operating systems is a threads-based virtual machine. A variety of mechanisms are provided for synchronizing threads; these mechanisms include locks, semaphores, and monitors. Likewise, mechanisms are provided for spawning threads, and for threads to terminate themselves.

Since these operating systems also support processes, we can think of the virtual machine for these systems as a threads/process based machine.

Windows NT

Windows New Technology (NT) is a full-featured environment developed by Microsoft as an upscale member of the Windows family [Custer 93]. This operating system was designed to be extensible, portable, robust, compatible with a variety of architectures, and high performance (i.e., fast and responsive). Particular attention was given to provide support for multiprocessor machines and features for advanced networking.

The Windows Open Systems Architecture (WOSA) is Microsoft's map for the future of operating systems. It provides support for many Applications Programming Interfaces (APIs), such as OS/2 and Windows 95. WOSA uses a Graphical User Interface (GUI), which features now-ubiquitous windows, as its main operating system for the user, and uses the command-line interface (e.g., DOS) as a behind-the-scenes tool. Windows NT appears atop the WOSA hierarchy.

Windows NT was developed with several requirements in mind. Specifically, the operating system is portable across a variety of architectures, and provides capabilities for multiprocessing. Networking features built into Windows NT allow for ease of distributed computing. An IEEE committee drafted a Portable Operating System Interface based on

UNIX [POSIX 90]; Windows NT is POSIX-compliant with its optional POSIX application execution environment. Additionally, Windows NT provides a DOD Class C2 Level of Security, permitting discretionary need-to-know protection and, through the inclusion of audit capabilities, allowing for the accountability of subjects and actions they initiate.

Windows NT uses the client/server model to provide multiple operating system environments (e.g., Windows, MS/DOS, OS/2, POSIX); an object model to manage operating resources uniformly and to dispense these resources to users; and a symmetric multiprocessing model to maximize performance. It is a multitasking operating system, desined to divide work among processes. Each process has memory, systems resources, and at least one thread of execution. The operating system handles thread scheduling.

Windows NT provides several synchronization objects that allow threads to synchronize their actions with one another. The four main synchronization objects are critical sections, mutexes, semaphores, and events; these and other details of the threads package is described in [Richter 94].

The Windows NT operating system manages memory, and regulates access to files and devices. Its structure consists of a user-mode portion and a kernel-mode portion. The user-mode (protected subsystems) allows each object system entity to reside in a separate process, protected from other processes by the kernel-mode's virtual memory system. Different processes communicate through message passing. The kernel-mode (NT executive) consists of many functional components:

- *Object manager* - handles creation, management, and destruction of objects.
- *Security reference monitor* - handles object protection and auditing.
- *Process manager* - handles creation, suspension, resumption, and termination of processes and threads.
- *Procedure call facility* - handles message passing between clients and servers, both locally and remotely.
- *Virtual memory (VM) manager* - provides a private address space for each process, protects each process's address space, and handles paging when memory usage is too high.
- *Kernel* - responds to interrupts and exceptions, schedules threads for execution, synchronizes activities of multiple processors, and supplies a set of elemental objects and interfaces.
- *I/O system* - handles device management, file systems, network redirection, network services, and cache management.
- *Hardware Abstraction Layer (HAL)* - provides callable routines for platform-dependent information usable by applications.

Mach

Mach is a modern, UNIX-compatible, microkernel-based operating system developed in the 1970s and 1980s [Rashid 86]. Its current primary goals, mentioned in an excellent overview of Mach in [Tanenbaum 95], are

1. Providing a base for building other operating systems, such as UNIX.
2. Supporting large sparse address spaces.
3. Allowing transparent access to network resources.
4. Exploiting parallelism in both the system and the applications.
5. Making Mach portable to a larger collection of machines.

What makes Mach useful is that few multiprocessor systems other than Mach are truly machine independent.

The Mach microkernel was built as a base upon which UNIX and other operating systems can be emulated by a software layer that runs outside the kernel; multiple emulators can run simultaneously, so it is possible to run 4.3 BSD, System V, and MS/DOS programs on the same machine at the same time. Like other microkernels, the Mach kernel provides process management, memory management, communication, and I/O services. The kernel provides mechanisms for making the system work, but lets user-level processes handle the policies driving files, directories, and other traditional operating system functions.

The Mach kernel itself manages five main abstractions:

- *Processes.* A Mach process is an environment in which execution can take place; it provides an address space for program text, data, and one or more stacks. Processes are the basic units for resource allocation (e.g., a communication channel is owned by a single process).

- *Threads.* A thread in Mach is an executable entity; it has a program counter and a set of registers associated with it. Each thread is part of exactly one process; a single-threaded process resembles a UNIX process.

- *Memory objects.* This concept is unique to Mach. A memory object is a data structure that can be mapped into a process's address space. Memory objects occupy one or more pages and form the basis of the Mach virtual memory system. When a process tries to reference a memory object not in physical memory, it gets a page fault, which the kernel catches. Unlike traditional operating systems, the Mach kernel can send a message to a user-level server to fetch the missing page.

- *Ports.* Message passing forms the basis for Mach interprocess communication. Ports are protected mailboxes created for receiving messages; a port is maintained inside the kernel, and it has the ability to queue an ordered list of messages.

- *Messages.* A process can give the ability to send to (or receive from) one of its ports to another process through a permission called a capability.

An excellent discussion of Mach process and thread management, memory management, communication, and UNIX emulation is provided in [Tanenbaum 95].

Solaris

SunOS's Solaris [Powell 94] provides an operating system also based on a model that allows multiple threads of control within a single process. Its main goal is to provide extremely lightweight threads, effectively extending the UNIX Application Programming Interface for a multi-threaded environment. The threads are sufficiently lightweight so that there can be thousands present, with synchronization and context switching accomplished rapidly outside the kernel; this feature is achieved by multiplexing user-level threads on top of kernel-supported threads of control. Such an operating system architecture allows the programmer to separate logical program concurrency from the required real concurrency, which is relatively costly, and to control both within a single programming model.

Solaris threads of control include a program counter and stack to keep track of local variables and return addresses. Programmers write programs using the threads package; a second tier of the Solaris system is the lightweight processes (LWPs), which is defined by the services the operating system must provide on the kernel to handle thread manipulations. Thread synchronization facilities include mutexes, condition variables, and semaphores; Solaris provides a uniform synchronization model between threads both inside and outside processes.

LWPs provide virtual CPUs that share address space; each LWP is dispatched and scheduled by the kernel. Threads are implemented using LWPs, which handle their scheduling and synchronization at the user level. The programmer can also explicitly handle the mapping of threads onto LWPs to achieve specific performance or functionality without leaving the threads model. In addition, the programmer can control the allocation of stacks and thread-local storage, which allows coexistence with other memory allocation models such as garbage collection.

A minimalist translation of the environment to threads allows higher-level interfaces such as POSIX Pthreads to be implemented atop SunOS threads.

OS/2 Warp

IBM's OS/2 Warp operating system allows prioritized multithreading of applications [Stagray 95, Reich 95]. It provides a workplace shell as its GUI interface. Like the other operating systems discussed in this section, OS/2's ability to perform tasks is based on the process/thread model. It provides power by permitting preemptive multitasking on a personal computer; that is, tasks can be assigned priority so that only the tasks specifically needing the CPU get it. This affords the programmer more control than the cooperative multitasking provided by DOS and Windows, in which all tasks have equal priority to use the microprocessor, even if they have no work to do. The efficiency of effective multitasking gives OS/2 application developers much control over their programs.

OS/2 was developed with speed of application execution in mind: it is a full 32-bit operating system, which means it can process 32 bits of information at a time. I/O routines are also optimized in the operating system: Lazy writing to the hard disk improves output performance, and the high-speed buffer allows for disk caching and read ahead. The operating system's kernel was developed to be robust, providing virtual memory management and garbage collection with a special provision for crash protection. The kernel itself contains the base file system, process and thread scheduler/dispatcher, exception handler, and memory manager. Applications developed for OS/2 often make use of Dynamic Linked Libraries (DLL), which are binary code components that can be shared by multiple applications and reused in new application design.

Operating Systems for Multicomputers

A key question for multicomputer operating systems is: Should there be a full operating system on each node of the multicomputer, or should there be only a small kernel at each node? In other words, should each node be a full computer? Since multicomputers such as the IBM SP2 are often used as computational servers, with some jobs requiring only one node, these multicomputers have complete (or almost complete) operating systems in each node. System-wide facilities are provided for job scheduling, for booting up, and for other system maintenance activities. Many scheduling heuristics have been studied [Ousterhout 82, El-Rewini 94].

The virtual machine layer provided by a multicomputer operating system is a process-based virtual machine. Since many applications are Single Program, Multiple Data (SPMD), multicomputer operating systems support gang-based scheduling of processes; a collection of processes (often one process to a processor) is scheduled as a unit.

The message-passing layers (e.g., Message Passing Interface (MPI) [MPI 93], Parallel Virtual Machine (PVM) [Sunderam 90, Geist 92], and Intel's NX Library) supported by multicomputers were designed to execute SPMD programs efficiently. Therefore, they support group communication between all processes. They also support point-to-point communication. It is helpful to consider three categories of virtual machines supported by multicomputer operating systems: (1) an SPMD model, with gang-scheduling of a set of processes using SPMD message-passing primitives, (2) a general process model with dynamic creation and deletion of processes and arbitrary communication patterns, and (3) a combination of the two with the SPMD model supported on top of the general model.

DCE

The Open System Foundation's Distributed Computing Environment (DCE) provides a distributed operating system built on top of existing operating systems (e.g., UNIX, VMS, Windows, and OS/2). Its primary goal is to provide a coherent, seamless platform for running distributed applications, without disturbing existing (nondistributed) applications. DCE offers many tools and services, plus an infrastructure in which they can operate; an example of such a tool is a mechanism for synchronizing clocks on different machines, constructing a global time clock.

As described excellently in Case Study 4 in [Tanenbaum 95], DCE runs on many different types of computers, operating systems, and networks. Consequently, application developers can easily write portable software that runs on a variety of platforms, reducing the costs of development. The distributed system on which a DCE application runs can be heterogeneous, and DCE was designed to internetwork with many different standards; the DCE software layer makes individual machine configuration differences transparent to the programmer.

DCE is based on a client/server model, in which user processes are clients that access remotely offered services. DCE itself provides many distributed time, directory, security, and file system services, as well as a threads package and Remote Procedure Call (RPC) provisions. The threads package furnishes a standard thread interface across systems, either using an operating system's native thread package, or providing one from scratch. RPCs provide communication in DCE, allowing client processes to call a procedure on a remote machine.

The DCE threads package is P1003.4a POSIX standard, and provides a library of user-level routines that allow processes to create, delete, synchronize and manipulate threads. Threads behave as discussed earlier; special attention was given to minimize the impact on existing applications. Thread scheduling is similar to process scheduling, except that it is visible to the application.

RPCs as provided by DCE allow much interface hiding, so that client and server processes can interact without knowing about specific implementation details. DCE's time service represents global time not as single values, but as intervals. DCE's directory service is a distributed database system that stores the names and locations of all sorts of resources, and allows clients to look them up in the name hierarchy (which supports both DNS and X.500). The security service in DCE allows authentication of clients, servers, and RPCs. DCE's distributed file system provides a system-wide name space for file access. DCE provides many facilities and tools, but it is not yet complete.

Operating Systems for Hierarchical Memory Machines

Each node of a hierarchical memory machine has its own operating system with some message layer for communicating between processors. It is possible, that a single operating system that spans a hierarchical memory machine will be developed. The virtual machine available today is a process/threads based machine, with an underlying message layer (e.g., TCP/IP) for communicating between processes.

Operating Systems for Geographically Distributed Memory Machines

Many different kinds of devices can be used in C4I systems; these devices range from handheld Personal Digital Assistants (PDAs) to supercomputers running simulations. The variety of devices leads to a situation in which subsystems have operating systems, and subsystems communicate using a message-passing layer. The virtual machine provided by the operating system layer for C4I systems is therefore a collection of different virtual machines and a communication fabric. This situation is likely to continue as even more different kinds of devices are added to the C4I fabric.

Operating Systems for Real-Time Systems

Operating systems for real-time provide mechanisms for fine control of process scheduling. The virtual machine layer is a process layer with the ability to define repeating tasks, deadlines and a variety of scheduling mechanisms.

Summary

The virtual machines provided by the operating system layer are, for obvious reasons, closely related to the virtual machines provided by the hardware architecture. We shall see that they are also closely related to the programming language and application layers.

A simple categorization of virtual machines at the operating system layer is as follows. Is it process based, or thread based, or both? What is the communication fabric; that is, does it provide point-to-point communication as well as more structured forms of group communication? And, does the virtual machine have facilities for specifying cyclic tasks and real-time deadlines and scheduling?

PROGRAMING LANGUAGE LEVEL ISSUES FOR VIRTUAL MACHINES

A large amount of research has been devoted to the design of languages that can be implemented on heterogeneous architectures and thus be treated as candidate virtual machines. In this section we survey the various programming paradigms that have been used to program parallel architectures, survey different languages, notations, and libraries that have been developed, and include a detailed discussion of some of the projects that appear to have the most relevance to the goal of designing virtual machines for C^3I applications.

The Cost-Benefit Model: Yardsticks for porting across Programming Languages

When considering a programming language port, the costs and benefits of designing using virtual machines can be compared using the following model. These tradeoffs must be carefully weighed to determine the suitability of using a virtual machine for this layer of a given project. The following questions are relevant for application developers when considering different programming languages:

- What are the architectures and operating systems being considered?
- Is the language sequential or parallel?
- If parallel, is the language explicitly parallel, implicitly parallel, or a hybrid?
- If explicitly parallel, is the language task parallel, data parallel, or integrated task and data parallel?
- If implicitly parallel, is the language functional, logic, dataflow, or collection-oriented?
- What language constructs are provided?
- What efficiency considerations are given by the language to the architecture?

Available parallel languages and libraries can be classified along numerous dimensions: implicit or explicitly parallel languages, imperative or functional or logic programming notations, distributed or shared-memory programming languages, universal or distributed address space languages and so on. Henceforth we will use the term notation to mean both languages and libraries. We categorize languages initially on the basis of the programming paradigm that they present to the user and subsequently address the architectural issues. Some of the most common paradigms that have been used to design notations for parallel architectures include the following:

- Sequential imperative languages with automatic parallelization

- Explicitly parallel imperative notations
    - task parallel (e.g., Ada [Ada 90], PVM [Sunderam 90], Maisie [Bagrodia 94], CC++ [Chandy 93], Linda [Carriero 89, Hasselbring 93], FORTRAN M [Foster 95])
    - data parallel (e.g., HPF [Koelbel 94, Foster 94c], DINO [Rosing 91b], pC++ [Bodin 91], C* [Hatcher 91b])

    - integrated data and task parallel (e.g., UC [Chakravarty 90], MPI [MPI 93])

- Implicitly parallel languages
    - functional (e.g., FP [Walinsky 94], SISAL [Feo 90], Haskell [Hudak 90], Crystal [Chen 88a, Chen 88b], Lisp [Zorn 89])
    - logic programming (e.g., Prolog [Ferguson 81, Warren 87], FCP [Shapiro 89], GHC [Tanaka 86, Saraswat 87a], Parlog [Gregory 87])
    - Dataflow languages (e.g., Id [Nikhil 88])
    - Collection oriented languages (e.g., SETL [Dewar 79, Freudenberger 83], Paralation Lisp [Sipelstein 91])

- Hybrid notations (e.g., PCN)

A number of parallel languages and notations have been designed within the imperative paradigm. Many languages were tailored to a particular parallel architecture, as was the case with Occam [Inmos 88] for the Transputer, C* [Thinking 90] for the Connection Machine (CM), and Cedar FORTRAN [Guzzi 90] for the Cedar multiprocessor. Other languages were devised for a family of architectures: Ada, Linda, Maisie and MPI are good examples of languages for asynchronous architectures. These efforts represent a large and significant body of research that has considerably advanced the state of art of parallel programming. In the domain of imperative languages with explicit parallelism, the primary research focus in the last few years has been on data parallel, integrated task and data parallel notations, and parallel object oriented languages. The former are discussed in this section while objects are dealt with separately in a subsequent section.

Explicitly parallel languages differ primarily in the manner in which they attempt to support the following facilities:*computation* model, *communication* model, and *synchronization* model. The computation model of a language refers to the language constructs used to decompose a computation (or program) into concurrently executing units. Commonly used computation models include the sequential model, data parallel model, task parallel model, functional model etc. The synchronization model of a language refers to the mechanisms used to synchronize the execution of the computation units. Commonly used models include the implicit barrier synchronization provided by data parallel languages and the explicit synchronization primitives provided by traditional task parallel languages. The two most common communication models are the universal addressable memory model where any concurrent module may directly reference any variable and the local memory model where private and shared variables are referenced using separate mechanisms. Languages with the universally addressable memory model additionally include data distribution models to optionally describe ways in which the program data structures may be mapped on the memory hierarchy of specific parallel architectures.

In an effort to implement languages with implicit parallelism on diverse parallel architectures, researchers have attempted to implement sequential (e.g., FORTRAN), functional (e.g., Crystal) or logical (e.g., Parlog) programming notations on heterogeneous architectures. Logical and functional notations are natural ways of expressing many computations and are excellent vehicles for software prototyping. However, it is difficult to refine programs in these notations to effectively exploit the architecture, while remaining within these programming styles. Also the inherent reluctance of the user community to the use of these notations is an additional obstacle to their widespread acceptance.

Parallelizing Sequential Notations

A considerable amount of research has been devoted to the development of compiler technology to automatically parallelize sequential programs, particularly iterative constructs like the FORTRAN do loops. A number of commercial products have also been designed. Parallelizing compilers [Gajski 82, Padua 86, Callahan 88, Polychronopoulos 88, Saltz 90, Balasundaram 91, Das 91, Koelbel 91, Saltz 91, Zima 91] allow sequential programs to be automatically executed on parallel architectures primarily by transforming sequential loops into a sequence of parallel or vector assignments. A number of sophisticated dependency analysis techniques have been developed [Ferrante 87, Banjeree 88, Banjeree 93] to allow a compiler automatically to identify the loops that can be parallelized.

Language Constructs

The primary approach is to identify the*data* and *control* dependencies in a program. A programmer imposes a specific control flow on the program using the control structures provided by the language; in most languages this imposes a total order on the operations executed in the program. However in most programs, many operations can be reordered to increase the number of operations that may be executed in parallel. The data dependence in a program refers to the orderings that must be maintained to preserve the semantics of the original program. This dependence is typically represented by a graph called the *data dependence graph (DDG).* The DDG is a directed multigraph whose vertices correspond to statements or operations in the program and an edge between two vertices corresponds to a constraint that prevents the corresponding statements or operations from being reordered. Some data dependencies are conditional; i.e. the dependence exists only under certain conditions. This is captured by using the notion of control dependence. Control dependencies are represented by using a program dependence graph which identifies the orderings among the basic blocks (single entry single exit) in the program. Informally a basic block*a* is control dependent on a basic block *b,* if the graph contains at least two paths from *b* such that taking one path means that *a* will always be executed and the other path may or may not result in the execution of *a.* A number of techniques and algorithms have been designed o construct these graphs for a given sequential program and reorder statements subject to these constraints to increase the degree of parallelism in the execution of the program.

Although originally devised for the analysis of single programs that operate on array data structures, these techniques have recently been extended to support interprocedural analyses as well as for use in programs that manipulate pointers, lists and other dynamic data structures.

Parallel Implementations

Efforts to parallelize sequential programs have primarily targeted vector architectures like the Cray X-MP and Y-MP series and shared-memory multiprocessor architectures. More recently these optimizations have also been applied to distributed memory computers.

For vector architectures, the dependency analysis is used to replace sequential iterations of inner loops in a program by vector operations with a stride determined by the size of the vector registers in the architecture. Subject to the constraints described in the previous section, the code is reordered such that memory (loads and stores) and arithmetic operations on individual elements can be replaced with corresponding vector operations.

For multiprocessor or UMA shared memory architectures, the dependency analysis is used to distribute the iterations of a loop among the processors such that the computational load is approximately balanced and the locality is improved to effectively utilize local processor caches. For implementation on distributed memory architectures, it may additionally be necessary to annotate the original program with data distribution primitives that indicate the manner in which the program data structures are partitioned on the memory hierarchy of the target architecture. Available facilities to specify the data distributions in a program are described in detail subsequently in the section on data parallel languages. Given a data distribution, each processor executes a sequential program and communicates with remote processors when it needs the corresponding data. Once again the dependency analysis is used to identify loop iterations that are independent and hence executable in parallel. The analysis is also used to generate the necessary communication and synchronization among the parallel processors.

Relevance to CVM

The primary strength of this approach is that it does not require programmers to learn new programming paradigms and it is possible to use existing sequential verification systems to prove correctness. However, its major drawback is that for many applications, it is impossible for the compiler to extract an efficient implementation. More recently, this model has been extended as FortranD [Fox 90, Hiranandani 91b] to include limited specification of data parallelism together with block SIMD like synchronization where communication is via a universal address space.

Another major drawback with this approach is that a parallelizing compiler cannot alter the algorithm that is programmed in the sequential notation. It is often the case that compared to the, often, sequential algorithm that is inherent in the program designed in a sequential notation, a more efficient parallel algorithms may be available.

This approach is applicable for scientific computations, particularly legacy code. However, it has limited applicability for the command and control functions of C^3I applications which often cannot be expressed in a sequential notation. Further, sequential notations are often a poor choice for prototyping purposes. (explicitly parallel programs for many typical applications are considerably shorter than equivalent sequential programs). Finally, novel architectures may suggest new algorithms for solving old problems as was the case with data parallel computations on the Connection Machine; parallelizing compilers are of little help in such situations.

Task Parallel Languages

The area of task parallel languages has been a rich and fertile ground for researchers and language designers over the last few decades beginning with the seminal work by Hoare on the notation called *Communicating Sequential Processes* (CSP) [Hoare 78, Hoare 84]. Most task parallel languages must provide constructs to

- *Specification of the units of parallelism:* processes, tasks or objects; static or dynamic process topology
- *Mapping of the units to the processors:* static or dynamic; user-specified or automatic
- *Communication mechanisms:* asynchronous message passing, synchronous message passing, remote procedure call, point-to-point message communications, one-to-many communications or shared memory
- *Synchronization mechanisms:* messages, semaphores, monitors, barriers

In addition, from the perspective of C^3I applications, it is also important to identify the following facilities that may sometimes be provided by the languages.

- *Failures:* detection or transparent fault tolerance
- *Atomic transactions:* atomic broadcasts, causal broadcasts, execute once semantics
- *Real-time characteristics:* ability to handle soft, hard, or semi-hard deadlines.
- *Performance prediction:* prediction or monitoring of performance for heterogeneous platforms

Language Constructs: Unit of Parallelism

The unit of parallelism, also referred to as *computation granularity*, in a language may be as small as an expression or be a larger unit like a program. The computation granularity is strongly related to the relative cost of communication versus computation for the architecture on which the language will eventually be implemented. Some languages require that the number of processes be fixed at compile time whereas most require them to be created and terminated dynamically. Real-time parallel languages, in particular, may specify a static set of processes for purposes of schedulability.

A *process*, sometimes called a *task*, is perhaps the most common unit of parallelism, and is used by a number of notations including Ada, CSP, Linda, Maisie, and MPI. Some notations use a *function* as the unit of parallelism. A process is a sequential program and contains local code and state. Typically the language provides the notion of a *process type* that may be used either to declare variables (implicit process creation) or be used to explicitly create processes using a*new* or *create* primitive. Most languages do not restrict the lifetime of a process and support autonomous process termination. A few languages (e.g., Ada) provide explicit constructs to abort other processes. A process, like a coroutine, may autonomously suspend its execution, and be resumed subsequently, for instance on the receipt of a message. This requires that each process is typically associated with its own stack and heap. Context switching from one process to another may thus be relatively expensive as it may require the stack, heap, registers and other state information to be saved.

A *thread*, sometimes called a *light-weight process,* is an alternative unit of parallelism. Like a process, a thread also contains local data and code. However unlike a process, a thread can share address space with another thread, such that context-switching overheads are much less as compared with processes. A number of thread packages are commonly available including DCE, Solaris, and Windows NT.

Parallel object-oriented languages use an*object* as their basic unit of parallelism. An object is similar to a process with two significant differences: first, an object encapsulates data and all code that operates on that data (called *methods*), making only selected data and methods accessible outside the object; second an object may be defined as an extension of a previously defined object by using*inheritance.* Parallel object-oriented notations are considered separately in a subsequent section.

From the perspective of CVM, objects that are implemented as light-weight threads appear to be the most useful unit of parallelism.

In the rest of the paper, we will use the term process in a generic sense to refer to the unit of parallelism used by any language.

Language Constructs: Process Mapping

Most languages treat a process as a logical unit of parallelism, rather than a physical unit. Thus multiple processes are typically assigned to a single processor. A process is typically assigned to a processor when it is created. The assignment may be specified explicitly by the programmer as is the case in languages like Maisie, or be deduced implicitly by the compiler or run-time system, as is the case in Ada. Programmer-specified assignment has the distinct advantage that the assignment can be made to balance the computational load and the communication among processes on different processors. Of course, this assumes that the programmer is knowledgeable about the communication and computation characteristics of the processes comprising the program which may not always be possible. Automatic assignment of processes may use a random assignment (e.g., Cantor [Athas]) or use an underlying set of heuristics that attempt to balance the computational load to select a unique processor. Once a process has been assigned to a processor, most existing parallel languages do not support dynamic task migration. Dynamic migration has two components: a *policy* component that identifies a task that must be migrated form some source processor to a specific destination processor, and a *mechanism* that must implement the migration and ensure that all subsequent message are correctly routed to the new destination for the migrated process. A vast quantity of literature exists on the design of static and dynamic policies for load management. However, due to the typically high cost of most mechanisms, dynamic process migration is not widely supported in existing parallel notations. Note that if a notation supports light-weight threads, explicit thread destruction and creation on a remote processor may be a more effective way to accomplish load balancing in a computation.

The scheduling of the multiple processes that are mapped to a processor is another important design factor in a language. The scheduler must solve two related problems.

- *Which process to schedule*: At any given time, more than one process may be ready for execution. Some languages (e.g., Ada) use static task priorities, whereas real-time languages may use deadline-based schedulers. Most parallel languages allow the scheduler to non-deterministically select any ready process.

- *When to suspend a scheduled process:* Some languages (e.g., Maisie) support auto-scheduling, where a process suspends itself autonomously. Although auto-scheduling minimizes context switching overheads, it may block and in the degenerate case starve other processes. An alternative is to use round-robin scheduling that may potentially increase context switching overheads but may lead to better overall performance by reducing the blocking time for a process; by adaptively varying the quantum of execution time that is allocated to each process, it may be possible to generate an efficient schedule using the round-robin strategy.

For C^3I applications, a task parallel CVM with a round robin, priority scheduler and a programmer specified process to processor mapping would be appropriate. Although facilities for dynamic task migration will be useful, it is unclear that a reasonable policy can be implemented efficiently. Thus it may be desirable to use dynamic load balancing only to correct severe load imbalances by explicitly terminating processes on the overloaded processor and creating them on the underloaded ones.

Language Constructs: Message-Passing Communication Mechanisms

A large number of languages use message passing as the primary means to communicate among the multiple processes in a program. In its simplest form, communication is achieved between two processes when one of them *sends* a message that is *received* by the other process. The most common form of message communication involves two processes, although many languages also provide communication constructs that may involve an arbitrary number of processes. A host of issues are involved in the design of these notations. Many surveys and books [Andrews 83, Bal 89, Andrews 91] are available that have treated the diversity of these issues in an exhaustive manner. Our goal here is to summarize the most popular forms of communication and examine their viability in a CVM for C^3I applications.

- *Active vs. passive messages:* In almost all existing languages, a message is a passive object. The receiving process contains the code that is to be executed on receipt of the message. Recent research has suggested the use of *active* messages, where the code to be executed for a message is 'carried in' the message. Active messages can significantly reduce the overhead of receiving and handling messages.

- *Naming:* some languages (e.g., CSP) use symmetric naming where both the sender and receiver respectively name the message destination and source; whereas some languages (e.g., Linda) communicate using a global mail-box (called a Tuple Space) where neither send nor the receive command names the source or the destination for the message. However the most common situation involves asymmetric naming (e.g., Ada, Maisie) where the sending process names the destination, but the receiving process has an implicit source for its messages. Some languages require that the names used in the communication commands refer to processes; other common options include *channels* (e.g., Occam, FORTRAN M), *ports* (e.g., Argus), and mailboxes (e.g., Linda). Channels are used to connect exactly two processes, ports have one destination but multiple sources, and mail-boxes may have multiple sources and destinations.

- *Blocking:* Many languages provide a blocking receive operation, although some like Ada allow programmers to specify a time-out interval such that if the desired communication is not completed within the specified period, the process is automatically unblocked. The degree of blocking associated with the send command has led to three primary forms of message communications.
    - *Buffered message passing* (e.g., Maisie, SR [Andrews91]): the sender process is blocked only until the message has been delivered to the communication medium. This would cause the sending process to block if the communication buffer becomes full.
    - *Synchronous message passing* (e.g., CSP): sender process is blocked until the message has been received at the destination; and
    - *Remote procedure calls or binary rendezvous* (e.g., Ada): sender process is blocked until it has received a reply to its message.

- *Message selectivity:* Some languages require that a process accept messages strictly in the order they are received. Others allow a process varying amounts of selectivity in deciding the order in which incoming messages are accepted. In languages like Ada, a process may influence the order of message acceptances based on the message type and its local state. Other languages like SR allow a process to additionally use the contents of the message to specify this order. Perhaps the most general model of message selectivity is provided by the Maisie language which allows a process to accept multiple messages of the same or different types as an atomic action.

- *Nondeterminism:* Most languages are non-deterministic in the sense that if at a given point, the message buffer of a process contains multiple*acceptable* messages, the language semantics allow the scheduler to non-deterministically deliver any one of the messages to the process. However some notations like FORTRAN M and Maisie contain constructs to restrict the non-determinism in a program such that at any given point in its execution, a process can accept only a single message of a given type.

- *Number of communication partners:* Message communication in most languages involves exactly two processes; however a number of languages specify*broadcasts* (e.g., BSP [Gehani 84], ISIS [Birman 87]) as their primary form of communication. Other languages have been suggested that specify communication abstractions called *scripts* which can be used to define communication patterns among an arbitrary number of processes in the form of a template. The templates can be instantiated at execution time to provide the corresponding form of communication.

Language Constructs: Shared-Memory Communication Mechanism

Many languages have been designed that use read and write operations to a shared address space for communication among the processes. As each process may autonomously read or write to the shared memory, synchronization constructs must be provided by the language to ensure that a process writing a data item does not interfere with another process that may simultaneously be reading it. Early constructs to control interference among simultaneous access to a shared address space include locks, semaphores, conditional critical regions, monitors and others. The primary purpose behind each of these constructs is to support the notion of an atomic operation: an operation that cannot be interrupted by another process. In other words, if a process is executing an atomic operation, other processes may only observe the state of the computation either prior to or following the execution of the operation.

Most architectures support a primitive operation at the hardware level that can be used to implement larger granularity atomic operations. For instance, a test-and-set instruction available on many architectures can be used to atomically test if a variable is 1, and if so to set it to 0. Such an instruction can be used to directly implement a binary semaphore which can be used by a process to obtain exclusive access to shared files and complex data structures. Binary and counting semaphore may, in turn, be used to implement other synchronization constructs and mutual exclusion algorithms.

Shared-memory languages also provide constructs for communication and synchronization among an arbitrary number of processes. Global communication or broadcasts may be supported simply by using shared variables that are accessible to all processes. Many languages also support global synchronization operations called barriers: A process that executes a barrier instruction is blocked until all processes involved in the computation execute the same barrier instruction. Barriers can also be used to synchronize a subset of processes in a computation; many architectures provide hardware support for the efficient implementation of barrier instructions.

The preceding languages used a strong model of data consistency called sequential consistency [Lamport 79]. This model assumes that a shared variable was stored in a single location in the globally addressable memory and any replication or caching of shared variables in the processor cache was transparent to the programmer. This view of shared memory was prompted by the early generation of shared memory architectures which used a uniform memory access or UMA memory model where all processors were connected by an interconnection network to a common memory module. The sequential consistency model ensured that the result of executing a shared-memory parallel program would be consistent with the execution of that program on a sequential architecture. In other words, any execution of the program on a parallel architecture could be mimicked by an appropriate scheduler on a sequential architecture.

However because this model could not be implemented efficiently on scaleable shared memory parallel architectures, alternative forms of consistency, called weak consistency, were proposed. The primary difference between the sequential and weak consistency models is that with a weak consistency model, execution of a parallel program can lead to program states that are unreachable in any execution of the corresponding program on a sequential architecture. There has been insufficient exploration of weak consistency models in the design of programming languages with most existing languages using a sequential consistency model.

Parallel Implementations

Task parallel languages have been implemented on sequential, shared memory and distributed memory architectures.

The primary problem in multicomputer implementations of message passing languages are process creation and naming, efficient message transmissions, and reliable message delivery. Most implementations use an underlying transport protocol that guarantees reliable message delivery such that this is no longer of concern to the implementor. Efficiency of a parallel programming language is perhaps most impacted by the message passing overhead in the system. Research in the design of interconnection networks in multicomputers has led to considerable improvements in the hardware latency of message communications; however reductions in software latency have not kept pace. The primary bottleneck is the necessity to use the operating system in transferring a message from the network to the process and vice-versa. Recent research in the use of *active message*s and related techniques has shown that it is possible to reduce the software overheads by use of techniques like active messages that dramatically reduce the context-switching overheads of message transmissions. Message passing languages may also be implemented on shared memory architectures by emulating the channel between two processes by memory that is shared among the corresponding processes.

Shared-memory languages can be implemented efficiently on multiprocessor UMA architectures. However efficient implementations of sequential consistency models on

NUMA architectures are harder. NUMA architectures must replicate the shared data to reduce the access time. Two types of operations can cause significant slowdowns: invalidation of replicated data when it is modified by a processor, and fetching remote data. To achieve efficiency, these operations must be overlapped with other computations (e.g., issue a non-blocking prefetch for data before it is needed) or aggregated with similar operations (e.g., pipeline invalidations or aggregated prefetches). Unfortunately there is very limited opportunity for exploiting these optimizations without violating sequential consistency semantics. Contemporary architectures like the DASH machine prototyped at Stanford use weaker consistency mechanisms like*release consistency* to increase the opportunities where these optimizations can be applied. A significant amount of research has also been devoted to using software techniques like distributed shared memory and global shared memory [Bennett 90a, Bennett 90b] to implement shared memory languages with weak consistency on distributed memory architectures.

Relevance to CVM

Explicitly parallel languages have been used both in the design and in the implementation of numerous C^3I systems. Most parallel machines provide some form of a message-passing or shared-memory notation that is designed to exploit the architectural idiosyncrasies of the corresponding architecture and which may or, as is more often the case, may not be portable to other architectures. Interest in devising a CVM was promoted primarily because of the wide disparity among the available explicitly parallel notations.

Considering the existing set of parallel programming tools, perhaps the closest candidate for a CVM is the *Message Passing Interface* (MPI) which is emerging as a de facto message-passing standard. MPI is a complex system that includes more than a hundred functions in its library, each having numerous parameters. Included functions support many of the task and data parallel constructs: specific libraries are provided to send and receive messages, global communication and synchronization operations including broadcasts, reductions, and barriers, data movements including scatter-gather, asynchronous query operations, and facilities to dynamically create groups of communicating processes. Bindings to MPI have been specified with many common imperative languages including C and FORTRAN. Primary drawbacks to the adoption of MPI as a virtual machine for C^3I applications is the lack of support for real-time computing, specification and visualization. Whereas MPI would be a suitable CVM for the programmer interested in implementing a system on a given architecture, it is perhaps less suitable for the designer and the systems analyst interest in specifying the applications.

Data Parallel Languages

Data-parallel languages use the universally addressable memory as their communication model, an implicit barrier synchronization, and typically provide optional data distribution primitives to specify how the program data is distributed over the memory hierarchy of the parallel architecture. The primary difference among the available data parallel languages arises in the granularity of its synchronization barrier.

Language Constructs

In most existing language proposals, the granularity of synchronization is determined by the specific constructs provided by the language. For instance, at one extreme we have essentially SIMD languages like C* and ParallelDistributedC where the synchronization is defined at the level of individual operations: all parallel threads are either executing the same operation or are idle. On the other hand, languages like Kali [Mehrotra 91] restrict access to remote data only at selected points in a program, like at the beginning of successive iterations of a *for* loop. This ensures that a Kali program needs to be synchronized only at precisely those points where communication is possible; the multiple threads can execute asynchronously between two successive synchronization points. Other languages like UC [Bagrodia 95] and DINO permit synchronization with multiple grain size: for instance parallel functions use copy in/copy out semantics so that they need to be synchronized only at the point of call and return; other statements are synchronized at a finer level of granularity.

The synchronization granularity specified by a program determines the computation model presented to the programmer and the efficiency with which the languages can be implemented on asynchronous architectures. In general, languages that are synchronized at the expression or operation level have relatively simple *sequential* semantics, as the global state of the program is known prior to the execution of any operation. However, implementing such a strict notation on an asynchronous distributed memory architecture can, in general, be expensive as a barrier may potentially be required before the execution of every operation. This is particularly the case if the language uses a universal shared memory model where no syntactic distinction is made between local and remote data. In contrast, a language with a weaker synchronization model forces the programmer to restrict remote data access to specific points in their code. In other words, this places the burden of maintaining a coherent view of the shared data explicitly on the programmer. However, an implementation of this model on an asynchronous architecture requires a smaller number of barrier synchronizations as compared with the previous model. Reducing barrier synchronizations improves execution efficiency by reducing both the blocking and communication times.

Languages like UC support synchronizations at varying levels of granularity from expression-level, block-level to function-level synchronizations. By supporting synchronization at multiple levels of granularity, UC supports an iterative approach to program design where an initial program can be designed using expression-level synchronization and can subsequently be refined for efficient implementation on asynchronous architectures.

The primary constructs provided by data parallel languages are: parallel operations (e.g., assignments, function calls, and loops), parallel data combinations (e.g., reductions, parallel prefix, and gather/scatter), and data placement (e.g., alignment and decomposition). Parallel data operations may be specified implicitly as operations on aggregate data structures like arrays and sets or explicitly. In this section, we only examine aggregate data structures like arrays and sets or explicitly. In this section, we only examine explicit operations and treat implicit operations on aggregate data items in a subsequent section.

Language Constructs: Parallel Data Operations

Parallel assignments are the simplest form of a parallel data operation. Most languages like UC and HPF specify parallel operations only on regular data structures like arrays. Other notations like NESL [Blelloch 92] and pC++ also support these operations on irregular data structures like trees and sets. The HPF parallel assignment has the following form:

FORALL *(range, ... range, mask) assignment*

where each *range* denotes a range for some identifier in the form*id = lower bound : upper bound : stride* and *mask* is a Boolean expression that masks out the values from each range for which the mask evaluates to false. The following statement computes the reciprocal of all non-zero elements in a 100*100 array called*a.*

FORALL *(i=1:100, j=1:100, a(i,j) <> 0.0) a(i,j) = 1/a(i,j)*

The UC parallel assignment statement is a significant generalization of the FORALL statement as the mask may contain any Boolean predicate that does not have side-effects and the statement can be any C or UC statement including loops, conditional statements or nested par statements. The UC statement has the following form:

par *(range, ... range*) **st** (*mask*) *assignment*

where the range can be specified as a declaration like

range *I:i = 0..9, J:j =I*

which declares *I* to be a range of values form 0 to 9 and declares identifier i as a dummy variable that may be used to reference any element of the range I; another range *J* is also declared which contains the same values as*I.* The preceding HPF statement may be written in UC as

par (I,J) **st** *(a[i][j] <> 0) a[i][j] = 1/ a[i][j];*

As parallel assignments are synchronized at the expression-level, the right-hand side of the statement is evaluated, logically in parallel, for every unmasked element in the range, and the computed values are then assigned to the variables on the left-hand side, again in a synchronous manner.

Parallel assignments can also be used to move data in various patterns including broadcasts, scatter, and gather. For instance, in the following UC fragment, the first statement is used to broadcast *b* to all elements in array *a,* the second statement is used to scatter elements in the first column in array *c* to the first row of array *a.*

par (I,J) *a[i][j] = b;*
par (I) *a[0][i] = c[i][0];*

Language Constructs: Parallel Combinations

Reductions are the most common form of parallel combinations. A reduction returns the result of executing an associative and commutative operation (e.g., addition or multiplication) to a set of values. HPF provides a number of reduction operations as intrinsic functions in the language. In contrast, UC provides a reduction operator that has the following form:

$op (range **st** *mask exp)*

The following example computes the sum of all positive numbers in vector *a*

sum = **$+** *(I* **st** *(a[i]>0) a[i])*

Commonly supported reduction operators include multiplication, addition, maximum, minimum, logical and, logical or. Some languages also provide library functions to implement other operations like parallel prefix.

Language Constructs: Data Distribution

Data parallel programs typically use a universally addressable memory model. However most parallel architectures are non uniform memory access or NUMA machines and the specific distribution of data on the memory hierarchy has a significant impact on program performance. Two primary forms of data distributions are used: *data alignments* and *data decompositions.*

Data decomposition refers to the distribution of (aggregate) data structures on the memory hierarchy. For instance, if the program operates on an array data structure, it must be partitioned with different partitions assigned to separate processors. Common decomposition methods for arrays include *striped partitionings* where each partition contains one or more contiguous rows or columns of the array, and *checkerboard partitionings*, where each partition contains a sub matrix, typically the same shape as the original matrix. The partitionings may be *block* or *cyclic.* In the former, a contiguous block of rows (or columns) from the original matrix is assigned to each processor; in the latter, contiguous rows (or columns) are assigned to the processors in a cyclical manner. Further, the partitions may be *uniform*, where each partition contains the same number of elements, or *non-uniform,* where the count may be unequal. HPF provides a number of pre-defined partitionings in the form of templates that may be used by the programmer;

alternately the programmer may also specify their own templates to partition program data structures.

Data alignment is used to specify the placement of a data structure relative to another; for instance, consider a program that uses a vector *a* of *n* elements and a matrix *b* with *n* columns. In general, the vector may be aligned along any row of matrix*b;* however the communication pattern in the program may be such that communication costs are minimal when the vector *a* is aligned with the first row of matrix*b.* Although research is in progress to analyze the communication structure of a program to extract efficient data partitionings, most existing languages either rely on programmer-specified partitionings or use very simple heuristics to partition the data structures.

Implementation on Parallel Architectures

Data parallel languages have been implemented on both message passing and shared memory architectures. Compilers for data parallel programs compile the source program to an SPMD program which typically includes one process (or thread) for each processor. Each statement of a process is executed using the *owner computes rule*: a statement is executed on the owner processor, where the owner of a statement is the processor whose memory unit contains the variable that is modified by the corresponding statement. The SPMD program must be synchronized using global barriers at the synchronization granularity specified in the source program. Dependency analysis techniques may of course be used to reduce the number of barriers that are actually needed in the implementation [Quinn 94]. Efficient implementations of these statements require that it be possible to identify the communication pattern implied by the assignment as early as possible, preferably at compile time. For instance for the HPF FORALL statement, the basic approach adopted is to use pattern matching techniques [Li 91], to classify the type of communication in the statement, and use optimized communication routines to implement the corresponding pattern. In general the communication pattern in each statement is said to be either *structured* or *unstructured, where* the former refers to statements where the source and destination processor for each communication is deducible at compile time; unstructured communications are those where this information can only be extracted at run time. Languages like HPF and UC have been implemented on distributed memory machines and are also being implemented on shared memory architectures. The primary difference in the two implementations is that the former must use data mappings that are specified by the programmer, whereas it is easier to extract the necessary information at run time for shared memory implementations without large performance degradations.

Relevance to CVM

Data parallel languages are an excellent vehicle for the programming of a large number of scientific applications, particularly for dense matrix applications. The recent research emphasis in this area has shifted to the solution of sparse systems using related techniques. Research and prototype compilers for data parallel languages like HPF and the newly emerging high performance C++ are becoming available and commercial products are likely to begin to emerge shortly. The primary drawback of these notations forC^3I applications is that they cannot be used to express the control and coordination aspects of these applications. Thus the primary utility of data parallel languages is likely to be for programming scientific computations within theC^3I framework. HPF or even MPI may thus be a suitable candidate CVM for scientific applications.

Implicit Parallel Notations

Logic and Functional Languages

In contrast to the imperative paradigm which is closely modeled after the von Neumann-like architecture of computers, functional [Hudak 89] and logic [Shapiro 89] programming languages take a more declarative view of program design. In contrast to imperative languages that compute using side-effects via assignments or pointer manipulations, the primary computational unit in a functional language is a function call. In pure functional languages, the value returned by a function depends only on the input parameters specified to the function and has no side-effects. Logic programming languages are declarative notations which define*predicates* using other predicates and logical operators.

Logic programming languages provide three types of potential parallelism:*stream* parallelism, and the more commonly available*and-* and *or-parallelisms.* Stream parallelism is available when the goals of two or more clauses share a variable such that one of them functions as a 'producer' process and the other as a 'consumer' process. However this approach presents only limited potential for speedup using parallel execution. AND-parallelism exists in a program that defines a clause as a conjunction of other clauses all of which must succeed for the clause to succeed. OR-parallelism can be used if a program contains multiple clauses for a given predicate that can be evaluated in parallel until one of them succeeds. In the following fragment, AND-parallelism can be used to evaluate the sub-goals Y, Z and P of clause no.1 in parallel, whereas OR-parallelism can be used to simultaneously evaluate both clauses in parallel.

1. X :- Y, Z, P.
2. X:- A, B, C.

Languages that support the preceding forms of parallelism include Flat Concurrent Prolog (FCP), Parlog, and Guarded Horn Clauses (GHC). Each goal in the language can potentially be evaluated in parallel and represents the basic unit of parallelism in these languages. All communication in parallel logic languages is via shared variables passed as parameters to goals; processes representing each goal synchronize by suspending on unbound shared variables. Languages like Concurrent Prolog use guarded clauses, where a clause is optionally preceded by a conjunction of predicates; the clause is evaluated only if the preceding predicates are satisfied. As the guards may themselves contain goals this can create an arbitrary level of nesting and also lead to problems where the evaluation of a guard has a side-effect but the computation is subsequently canceled due to backtracking.

Other languages like FCP disallow user-specified predicates in a guard to ensure that the guards do not have side-effects and do not lead to arbitrary levels of nesting. Although many versions of these languages have been implemented on general purpose parallel computers, none of the implementations can yield performance for large applications that is comparable to imperative parallel languages.

In functional languages the primary unit of parallelism is the function; multiple arguments of a function call can be evaluated in parallel, which is useful for architectures that can exploit fine-grained parallelism. Arguments are typically evaluated using lazy evaluations. New function calls can be assigned to specific processors automatically by the run-time system or via explicit programmer annotations.

Relevance to CVM

Parallel logic and functional languages are excellent vehicles for program prototyping as programs in these notations are typically a few orders of magnitude smaller than equivalent programs in imperative languages. However almost no efficient implementations of these notations exist for contemporary parallel architectures and relatively little research has been devoted to efficient compilation of these notations into imperative languages (with the exception of the work on Strand [Foster 90a] and PCN [Foster 90b]). Finally, in spite of decades of research into different forms of these languages they have made almost no headway in popularizing their adoption by the C3I applications development community or by the larger programming community. These notations may be suitable as CVMs at the specification layer, but perhaps more abstract notations including temporal logics may be a better option at that level.

Miscellaneous Notations

A number of parallel languages have been devised using the notion of sets, relations, and other aggregate data structures. Some of these languages are explicitly parallel whereas others tend to specify the parallelism implicitly.

SETL is a high-level prototyping language that uses dynamic, heterogeneous sets as its primary data type. Control-flow is specified in SETL using the standard sequential constructs; presumably parallelizing compilers may be used to parallelize SETL programs much like they are used with FORTRAN loops, although the dynamic typing may make compile-time dependence analysis less effective for SETL loops. A parallel variant of SETL called Parallel SETL has been proposed recently. Parallel SETL attempts to exploit parallelism in a SETL program by explicitly specifying parallel executions of loops over both ordered and unordered collections. Using different keywords, the language can specify simultaneous SIMD or MIMD execution of multiple iterations of a loop that roughly corresponds to using the UC **par** or **arb** composition operators. The language also provides a standard par-begin/ par-end construct to specify asynchronous execution of a set of statements; however all communication requires global synchronization. Another prototyping effort is PROSET that integrates the generative communication constructs of Linda into SETL to design a task-parallel variant of SETL with multiple tuple spaces.

Paralation Lisp is a data-parallel LISP-extension language. It is based on the notion of a *paralation* which is similar to the notion of a relation. A parlation is a set of related fields with a distinguished field serving as the*index* that is used to enumerate the other fields. A communication pattern in Paralation Lisp is created by matching values of elements in two fields of the same or different paralations. A single communication mechanism, called move, is provided by the language. The construct uses pattern matching to set up a communication pattern and subsequently causes the matched elements to be transferred. If a match has multiple sources or destinations, the corresponding communication works as a reduction or broadcast.

Real-Time Languages

The importance of the timing behavior of manyparallel systems, particularly C^{3}I applications, has led to the design of a number of parallel real-time languages. The timing characteristics fall into two broad classes:*soft* and *hard*, where the former refers to deadlines that can be violated on an individual basis but collectively must be satisfied, and the latter refer to absolute timing constraints which, if violated, would result in an erroneous program execution.

A parallel real-time language must provide constructs (implicit or explicit) to specify the units of parallelism and the mechanisms for their communication and synchronization. In addition, the language must provide construct to specify timing constraints on the execution of the programs. Systems that are not subject to timing constraints use a relatively simple scheduler which may non-deterministically schedule any one of a number of ready jobs. In contrast, schedulers for real-time systems use specific criteria to select the particular job that is to be scheduled next. Suggested criteria include static priority as used in Ada, earliest deadline, latency, or the notion of adaptive priority . Algorithms for dynamically scheduling real-time tasks that have hard deadlines have been proposed in [Zhao 87]. To support the specification of timing constraints, the programming language must provide appropriate scheduling information for each schedulable unit of work (referred to as an "activity") in the system. This information is used by the scheduler to schedule the activity so as to obey the overall timing requirements for the system. In process-based languages, each activity is assumed to be initiated by a message and the appropriate scheduling information is transmitted to the scheduler within the message.

A number of languages have been proposed to program real-time systems. Ada was initially proposed to program real-time embedded systems. It supports static task priorities, but due to the absence of constructs to specify timing constraints and other problems, it can only support soft real-time systems.

Lee and Gehlot introduced language constructs for timing specification, communication, and exception handling. In addition, Real-Time Euclid [Kligerman 86] was proposed to guarantee schedulability where dynamic features, such as recursion and dynamic process creation were disallowed.

These languages augment traditional distributed programming languages with constructs for scheduling processes and/or for specifying timing constraints. They are restricted to systems that can be modeled by the classical periodic and sporadic task models and are typically tied to a specific process decomposition and/or a specific scheduling strategy.

To program hard real-time systems, the language must provide facilities whereby stringent timing constraints may be specified for each individual activity. These constraints are then used by the scheduler to schedule the activities appropriately. We consider a language called RTM [Bagrodia 91] that uses the traditional notion of message-communicating processes to program parallel real-time systems. The smallest schedulable unit is the message rather than an entire entity. Therefore, computational requirements of a real-time system specified in an activity-level model can be transformed into a set of RTM entities without restricting the transformation to a specific decomposition strategy.

Depending on the nature of the application and the scheduling algorithm to be used, the timing constraints may be specified in different forms. Accordingly, RTM provides a general notation to specify the scheduling information associated with each message. The language defines a struct type called*type_scheduler_info* which consists of three fields: ptime which refers to the processing time needed for the activity,*deadline* which refers to the deadline for completion of the activity, and*priority* which indicates the relative message priority. A pre-defined attribute,*my_priority*, is used to refer to the priority of the sending entity. For the periodic and sporadic task model, the scheduling information does not change for subsequent instances of the same activity; however in the adaptive task model, it may be dynamic. If the scheduling algorithm uses a different set of parameters than those defined above for scheduling messages, the*type_scheduler_info* may be redefined by the programmer to include the appropriate parameters. The scheduling information is associated with a message rather than an entity, in order not to force a specific program decomposition on the programmer. The RTM scheduling constructs can be used to describe sporadic tasks, periodic tasks, and tasks with adaptive priorities.

Relevance to CVM

Typical requirements of real-time systems like the detection and recovery from errors, and the specification of timing constraints are an important component of CVMs for C^3I applications. Regardless of the syntactic form of the CVM, it is clear that constructs to specify timing constraints on specific system-level activities are essential. This implies that unlike parallel languages for, say, scientific applications, CVMs for C^3I applications must also include provisions for selecting specific scheduling algorithms in the implemented system. Incorporation of timing constraints is also important for CVMs at the specification layer. In [Shaw 89], a methodology based on an extended Hoare logic was proposed to specify and prove assertions about time in higher-level sequential languages. This approach may be used to verify timing properties of implemented systems. In Real-Time Logic, time is captured by the @ function which assigns time values to event occurrences. The start and stop events of each action capture the progress of real time for each action. A rapid prototyping approach has also been proposed by [Luqi 93] to design real-time systems. In this approach, the initial design of a real-time system is described in PSDL, a prototype system description language. The PSDL specification is subsequently translated into Ada code together with scheduling information (static and dynamic schedulers). The Ada program is then executed together with the scheduler to test the feasibility of the specified real-time constraints. The code for the prototype is generally not used in the final implementation because the prototype is not a complete representation of the final system.

Summary

A significant amount of effort has been devoted to the design and implementation of parallel languages. Many of the languages considered in this section have been implemented on two or more parallel architectures and may hence be likely candidates for CVMs. Of all the notations that have been devised, MPI is perhaps the most widely available notation and is a de facto standard for message passing. MPI also supports many of the data combining and synchronization features of data parallel programs. Its primary drawback from the perspective of C^3I applications is that it does not provide any support for time-constrained computations. Another deficiency is the absence of any type of performance modeling capability that may be used to predict the performance profile for an application on diverse architectures. Some compilers and languages now provide performance prediction capabilities: one example is the Maisie environment which combines a message-passing programming notation with a simulation capability that may be used to model a variety of parallel software and hardware environments.

HIGH-LEVEL NOTATIONS AS VIRTUAL MACHINES

In this section, we discuss the use of high-level notations as a form of virtual machine. Many excellent high-level notation development manuscripts exist [Chandy 88, Gibbons 93, van de Snepscheut 93, Foster 94c, King 94]. For virtual machine considerations, we discuss three fundamental high-level notation forms:

1. **Specification Notations**. A specification notation describes*what* a system does. In contrast to a programming language, a specification notation might not describe*how* a system operates. Specification notations need not deal with many of the details handled in a programming language; they need not even be executable. Therefore, obtaining an implementation from a specification can require much effort.

2. **Domain-Specific Notations**. A domain-specific notation is specialized for a domain such as avionics. A domain-specific notation can use terms and procedures that have clear definitions in the domain, but have little meaning outside that domain.

3. **Very High Level Programming Languages**. Some languages have high expressivity, but may have relatively inefficient implementations. For example, APL is an expressive language in array processing. Also, SETL is an expressive language for set operations. One can even think of Mathematica, Maple and MATLAB as expressive languages for symbolic mathematics. High level languages can be used for rapid construction of prototypes. These prototypes serve both as reference implementations, and as de facto specifications.

We provide cost-benefit analyses with each of these high level notations.

The Cost-Benefit Model: Yardsticks for porting across High-level notations

When considering a high-level notation port, costs and benefits of designing using virtual machines can be compared using the following model. These tradeoffs must be carefully weighed to determine the suitability of using a virtual machine for this layer of a given project. The following questions are relevant for application developers when considering different high-level notations:

- How does the notation handle the specification?
- What domain-specific features are included with the notation?
- What are the relative expressivity and efficiency of the notation?
- How can the notation be implemented on a given architecture, operating system, and programming language?
- Is the system being modeled reactive or nonreactive?
- How much formality is necessary in the high-level design?

Specification notations

A specification notation describes*what* a system does, but not necessarily *how* a system operates. Even though a specification notation may not be executable, it still can play a valuable role in virtual machines. Recall that the primary goal of virtual machines is to reduce the effort required to port an application from one architecture to another, and a secondary goal is to reduce the effort and time required to maintain and modify an application. Specification notations can be ideal in meeting both these goals.

There are several cases of applications written in "legacy" code that can only be ported to modern machines with extraordinary effort. Examples include applications written in languages that are no longer actively maintained in the COTS area (e.g., JOVIAL), and implementations in assembly or near-assembly language in machines that are no longer supported. The options for porting these applications are either to rewrite the application, or to implement an emulator for the application language. In many cases, implementing an emulator is the only viable option because the applications are complex and there is no documentation that describes what the application actually does. In essence, the problem is that there is no specification.

Emulation is not a long term strategy. Though emulation on a modern machine can result in substantial speedup and solve the efficiency problem in the short term, it does not address the critical issue of maintenance. The only long term solution is to understand what the implementation does. In other words, the only long term solution is to reengineer the code to obtain a specification of the application. The effort that goes into this reengineering is orders of magnitude higher than the effort required to obtain a clear specification in the first place. Reimplementing the application in a different language, once a specification is obtained is feasible (though expensive). But, it is impossible to reimplement an application without understanding what it does. Therefore, specification notations can reduce the design effort required to port an application to new languages and architectures, and thus play an important role in virtual machines.

From the point of view of virtual machines, an important issue is to have a standard specification notation. The notation can be formal or informal, but it is essential that it be standard, so that the same terms mean the same thing to all its users. A difficulty with informal notations is that they can be ambiguous, and it is difficult to enforce common standards. A difficulty with formal notations is that, for many programmers, implementing a specification may take much less effort than writing the specification in a formal notation, and this can lead to "specification by prototype." A problem with the use of prototypes for specification is that many design decisions are taken in developing a prototype, and using the prototype as the specification can result in over-specifying the program.

Specification of designs

Specification notations are helpful in specifying both a system and designs. An important step in designing a system is to postulate and to define the subsystems. Porting an application requires less effort given the specification and rationale for a design, and given the specifications of the objects used in the design. There are layers of specification-based virtual machines with more abstract system-wide specifications used at earlier stages of design, and more detailed subsystem specifications used later in the design.

Specifications of Reactive and Nonreactive Systems

Different types of specification notations are used for reactive and nonreactive systems. The fundamental difference between these two types of systems is that whereas reactive programs interact with their environments during execution, nonreactive programs do not.

Nonreactive Programs

A nonreactive program has its inputs defined when the program is initiated, and it produces an output that can be specified by a relation between its inputs and outputs. In addition, a nonreactive program does not interact with its environment during program execution. Hence, a nonreactive program can be specified simply as a pair of predicates on variables of the program: the *precondition* and the *postcondition*. The specification requires that if the precondition holds when the program is initiated, then the program will terminate in a state in which the postcondition holds.

Nonreactive programs can also be specified with a predicate between the values of variables before the program is initiated and after the program terminates. For instance, using the notation *x.pre* and *x.post* respectively for the values of x before and after the program is executed, a program that can modify x and increases its value by 1 can be written as:

modifies x and x.post = x.pre + 1.

We note also that nonterminating programs can also be specified in terms of their weakest preconditions, but the critical issue here is not notation but the essential difference between reactive and nonreactive systems.

A nonreactive system can be implemented as a concurrent program. The definition of a nonreactive system does not require the system to be implemented as a sequential program; it only requires that it be specifiable by a relation between its input values and its output values. A performance-oriented virtual machine used for nonreactive applications on parallel computers, is Block Structured Programs (BSP). The program model is essentially that of a single thread that forks into multiple (often identical) threads that join again into the single thread. In effect, the program model is simply the sequential composition of parallel blocks. BSP programs can be implemented in a variety of programming languages.

Reactive systems

Reactive systems interact with their environments while they are executing. Reactive programs (e.g., operating systems) may not necessarily terminate execution. Therefore, it is not possible to specify reactive programs only in terms of the values of its variables before it begins execution and after it terminates execution. Reactive processes can be specified formally in terms of process algebras such as CCS , theoretical CSP [Hoare 84], and temporal logics such as Temporal Logics of Actions (TLA) [Lamport 94] and UNITY [Chandy 88].

A problem with specifying systems, particularly reactive systems, is that they are required to have pragmatic properties such as performance, fault tolerance, security and safety, in addition to their essential functional properties. Formal specifications of these properties can be very difficult. Specifications of graceful degradation of a missile launch detection system, for instance, can be extremely complex and is almost impossible without a specification of the design as well. There are pragmatic limits to using formal specification notations as virtual machines.

Temporal logic specifications have two essential components:*safety* and *progress*. The safety specification deals with system behavior that can be checked by inspecting finite behaviors. The most basic safety specifications deal with initial conditions and system state transitions: If one predicate holds in a given state, then another predicate must necessarily hold in the next state. Progress properties deal with infinite behaviors, and the most basic form of progress is: If a predicate holds at a point in the computation then another predicate will hold at a later point. Though temporal logics can deal with time, for the most part they deal with eventualities rather than Newtonian time. Many DOD systems, however, have to respond within limited periods of time. For such systems, the usual ways of defining progress in temporal logics can be inappropriate.

Cost-Benefit Analysis

Specifications are essential for portability. The costs of reengineering high-level specifications are so much higher than the cost of initially obtaining clear specifications that there is little question about the value of specifications. From the point of view of virtual machines, the critical question is: Should there be a standard specification notation that can serve as a virtual machine?

Today, the costs of enforcing a standard formal specification notation are very high and may exceed the benefits that derive from a formal standard. There are several reasons for the high cost of enforcing formal standards. There are no de facto formal standards in the literature and research community. There has been relatively little attention paid to formal notations for specifying properties such as fault tolerance and graceful degradation, and security and ability to withstand electronic warfare. There are few tools that support formal specifications of reactive systems.

In summary, the cost-effective approach at this point in time appears to be to enforce systematic specifications of systems and designs within standard programming practice, and to encourage standardization of specification notations but not to expect a complete formal specification standard, at the level of a virtual machine.

Domain-specific languages

The traditional way of viewing software development is that given a specification, develop an implementation. Some group, other than the development group, is responsible for program maintenance, and much of the maintenance effort is, in fact, continual modifications of the specifications so that the application has more functionality or is more efficient (or possibly both).

We now realize that the maintenance function is, in quintessence, a program and design *reuse* function: Given a program that satisfies a specification, and given modifications to the specification, develop a new program, reusing the old program and design as appropriate. We also realize that the initial design can also reuse even earlier designs and prototypes. These realizations have shifted emphasis from developing a single program to developing families of related software component products, with maintenance treated as part of the process of developing new products in this family.

This shift in emphasis has implications for virtual machines. If our mission is continuous evolution of products in a related family, a virtual machine that is especially suited to our product family would be useful. One such virtual machine is a programming language specific to a product family or specific to a domain.

The advantage of a domain-specific virtual machine is that developing programs for that virtual machine can require less effort. The disadvantage is that it requires a translator (e.g., compiler, interpreter, or human-translation methodology) from the domain-specific virtual machine to a widely-used virtual machine. Furthermore, this translator must be maintained. The critical tradeoff is the complexity of the translator and the complexity of the application.

Class libraries

A different approach to developing domain-specific virtual machines is to use domain-specific class libraries. The idea is to develop a "foundation class" library for a family of applications, and to build all the products by using the common foundation class, as much as possible. No translation mechanism is required.

A problem arises when a programmer attempts to enforce a discipline that reuses the common foundation class library as much as possible, instead of just starting from scratch each time and adding ever-increasing numbers of classes to the library. In other words, the problem is that of defining and truly using a virtual machine, and avoiding "creeping functionality" for the virtual machine.

A comparison of domain-specific class libraries and languages, from the viewpoint of virtual machines, is instructive. Languages offer the possibility of optimization based on domain knowledge; such optimization can be more difficult to obtain with class libraries. A language can also be better tailored to an application family because a class library is constrained by the underlying object-oriented programming system. On the other hand, languages require translators.

Cost-Benefit Analysis

As software development organizations shift their focus from developing sequences of software products to ongoing maintenance and enhancement of families of related products, the benefits of domain-specific virtual machines will dominate their costs.

The costs of developing a domain-specific virtual machine, using programming languages, specification notations or class libraries, are substantial. These costs payoff only if they can be amortized over the development of many products. Therefore, the critical issue in cost-benefit analysis is whether a given domain is important enough and focused enough that domain-specific effort will pay off. Costs of such efforts will outweigh benefits if the domain is too broad (i.e., if the domain encompasses all of C4I). The benefits will be too small if the domain is too narrow.

There are, however, domains that can be viewed as long term product families for DOD. These include target tracking, terrain masking, and trajectory optimization. The problem now is to extend work on Domain-Specific Software Architectures (DSSAs) [Hayesroth 95] to define virtual machines for these domains. This endeavor requires taking earlier work on DSSAs to a greater level of specificity. There appear to be domains with the appropriate degree of generality and the appropriate importance for DOD to make domain-specific virtual machines cost-effective for these domains.

High-Level Programming Languages

High-level programming languages can also serve as virtual machines. Since high level languages have been proposed for some time, we do not discuss the basic ideas underlying them, but go immediately to a cost-benefit analysis of such languages as virtual machines. Therefore, our focus is on the use of high-level languages to reduce the time required to port applications to new architectures and to maintain/enhance applications. We note that a very close relationship exists between high-level languages and domain-specific virtual machines, because most high level languages are designed to be particularly useful for limited domains and generally useful in all domains.

A high-level language can be used to build prototypes rapidly, and the prototype can serve as a reference implementation (even though it might be inefficient) or as an operational application in the field. Many high-level languages have the risk that they are not as widely supported as languages such as C, FORTRAN, Ada and C++. Therefore, porting an application implemented in a high-level language can be difficult a decade from now, because the language may not be adequately supported. Another cost is that reference implementations require several design decisions, and different decisions may be appropriate for an operational application; over-specification can carry substantial cost.

A critical problem with reference implementations is that they do not specify properties other than functional behavior; for instance, a reference implementation does not usually specify graceful degradation or resistance to electronic warfare because high-level languages are not equipped to deal with such issues. Therefore, reference implementations in high-level languages do not do away with the necessity of specifications.

A different kind of cost deals with interoperability. Many high-level languages are designed for specific domains (e.g., APL for arrays and SETL for sets). By contrast, many DOD applications span different domains. Therefore, for a prototype implementation of DOD applications, interoperability between high-level languages is important. Also, object-oriented languages offer the potential of high-level domain specific libraries (e.g., for arrays, sets, relations, and graphs), and with inheritance these languages and libraries offer much of the functionality of high-level languages without requiring special compilers or interpreters.

There are many benefits to using high-level languages. The constructs of high level languages can resemble the constructs used by people working in the field for which the language was designed. For instance, SETL deals with set union and intersection; these concepts do not need to be defined to a set theorist. Therefore, the virtual machine provided by a high level language can be close to the "virtual machine" used by people working in the field.

Also, high-level languages can be compiled and executed on an actual machine. Thus, potential consumers of an application can test a high level language implementation. Determining whether the specifications of an application are complete is very difficult, and testing a reference implementation can help in this regard. Maintaining applications implemented in high-level languages is simpler because the applications are more succinct and the relationship between specifications and code is more readily apparent.

Furthermore, compilers and runtime systems of high-level languages can carry out functions such as garbage collection, that are the programmer's responsibility in languages such as C and C++. Therefore, high-level programming languages can be of substantial value for porting and maintenance, and so they should be considered as one of the ways of implementing virtual machines.

Summary

Very high level languages offer one way of defining virtual machines. We have placed such as languages into two broad categories: executable and nonexecutable. Specification languages may be nonexecutable and have certain advantages, and executable high level languages have other advantages.

We also considered two other categories: domain-specific or general, and discussed these categories with respect to virtual machines. Given the state of research today, there are significant costs with defining formal virtual machines using high level notations; nevertheless, they can play an important role in developing the virtual machine concept.

INTEROPERABLE OBJECTS AS VIRTUAL MACHINES

A virtual machine layer above the programming language layer is the interoperable object and process layer. This layer encapsulates computational entities in binary form so that they can operate with each other (i.e., *interoperate*). The virtual machine defines a standard interface for objects, and an Interface Definition Language (IDL) is provided for conversion among different interfaces, to a common interface. This virtual machine layer is among the most important new technologies developed in the last few years.

We note that interoperable objects represent the culmination of both object-oriented technology research in both the sequential [Rine 95] and parallel [Agha 94] realms, and the spirit of design patterns [Gamma 95] for reusable object-oriented software.

The Cost-Benefit Model: Yardsticks for porting using Interoperable objects

When considering a port using interoperable objects, the costs and benefits of designing using virtual machines can be compared using the following model. These tradeoffs must be carefully weighed to determine the suitability of using a virtual machine for this layer of a given project. The following questions are relevant for application developers when considering interoperable objects:

- How are objects created and destroyed?
- What are the mechanisms for full and one-way remote procedure calls?
- How is object naming achieved?
- How are handles to objects obtained?
- What mechanisms does the environment provide for processes and threads?
- What are features specific to the object model being considered?

Object and Process Layers

An interoperable-object virtual machine supports interfaces between computational entities, so the effort to translate between different specific interfaces and the generic interface is reduced. Procedures, with arbitrary scoping of data, are insufficient for this purpose, because all variables in the procedure's scope must be converted from a specific format to a generic format and back, when the procedure is called. Objects and processes, therefore, are more suitable for providing these interfaces. Communication between objects is achieved using Remote Procedure Calls (RPCs); communication between processes is performed using either asynchronous or synchronous messages.

Some discussion of processes, threads, and RPCs was given in the operating systems virtual machine layer section; here, we describe a virtual machine based on objects.

Object-Based Virtual Machines

The central questions for object-based virtual machines are how an object is created, named, and terminated. In addition, it is important to know how object name handles can be obtained, and how RPCs are invoked.

The critical issue for interoperability, for objects implemented in different languages, is RPC execution. When a thread in one object executes an RPC on another object, the parameters of the RPC are in the data format of the calling (client) object. They are converted through a common interface format into a message, which is sent to the receiving (server) object. This object unpacks the parameters, and converts them into the format it needs.

A thread is instantiated in the remote object, and the procedure is executed within this thread. When execution of the thread completes, the output parameters are converted from the format of the receiving object into the common interface format and marshaled into a message, which is sent to the sender object. The parameters are unpacked from the message, and converted from the common format to the format expected by the receiver. Then the values of the output arguments are assigned to the appropriate variables in the receiver. From the virtual machine point of view, these virtual machines allow binary objects to execute RPCs on each other.

When an object is created, it has a handle and a set of attributes. An RPC can be executed on an object identified by its unique handle, or any object identified by a generic type name or set of attributes. The Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) [OMG 95] has an object repository that permits an object to find another object with the desired attributes, and this can be done dynamically.

The object-based virtual machine is an extremely important virtual machine for several reasons:

- A very large number of companies belong to OMG, and many of them support, or plan to support CORBA.

- OSF's Distributed Computing Environment (DCE) has some of the features of object-based virtual machines, and the number of companies that support at least one of CORBA and DCE is very high.

- CORBA is a threads/process based virtual machine in the sense that an object is similar to a process in having an address space, and there can be multiple threads executing concurrently within an object.

- CORBA offers the potential of true interoperability across binary objects developed in different languages and implemented on different hardware platforms.

Interoperable Objects Survey

A special issue of *Dr. Dobb's Journal* [DDJ 95] contains excellent descriptions of interoperable object issues. An interoperable object is language-independent, and makes use of distributed computing and compound-document technologies. Presently, they are more of a goal, rather than an employable reality. We note the distinction between object models such as CORBA, IBM's System Object Model (SOM) [SOM 94], and Microsoft's Component Object Model (COM); and application-level technologies (discussed in the next section) such as OpenDoc (implemented over SOM), OLE (implemented over COM), and TalAE (implemented over SOM).

CORBA

OMG's CORBA is a still-emerging standard for computing with distributed objects. Current systems based on (or compliant with) CORBA include IBM's Distributed System Object Model (DSOM), NeXT's Portable Distributed Objects (PDO), DEC's Object Request Broker (ORB), and Sunsoft's Distributed Objects Environment (DOE), and in addition, many smaller vendor implementations of CORBA, (e.g., Iona) exist. OMG is a consortium of more than 300 hardware, software, and end-user companies, and revision 2.0 of the specification is available in [OMG 95].

The Object Management Architecture (OMA) of CORBA defines a broad range of services and relationships within an environment, as well as providing object and reference models. The specification also includes an Interface Definition Language (IDL) and Common Facilities (CF). CORBA was designed to be cross-platform and cross-operating system.

OMA's underlying object model has clients sending object-identifying messages to servers, with zero or more parameters in the request. The model strictly segregates interface and implementation, so that the interface between components can be defined independent of their implementation languages. The *interface* combines the methods that can be called on an object, together with that object's accessible attributes.

By defining the interface, using the IDL, the application developer describes how the object appears to the Object Request Broker (ORB), which manages the communication interactions between client and server objects. The ORB's duties include all the responsibilities of a distributed computing operating system, from location and references of objects to the marshaling of request parameters and results. The ORB contains provisions for the dynamic invocation of RPCs on objects.

COM

Microsoft's COM is a component-software architecture that allows applications and systems to be built from components supplied by different software vendors; it forms the foundation for higher-level software services like those provided by OLE (e.g., compound documents, controls, interapplication programmability, data transfer, storage, and naming). In COM, an object is a piece of compiled code that furnishes some service to an application or the system; components are composed of one or more objects. COM is not a specification for how applications are structured: Rather, it is a specification for how applications interoperate.

COM defines a binary standard for component interoperability, is programming-language independent, is provided on many platforms (e.g., Windows, Macs, and UNIX), is extensible, and allows the robust evolution of component-based applications and systems. In addition, COM features mechanisms for communications between components (both across processes and across networks), error and status reporting, and dynamic component loading. COM enables application developers to deal with issues such as basic interoperability, versioning, language independence, and transparent cross-platform interoperability.

The design of COM has foundations in several fundamental ideas. COM provides a binary standard for function calling between components, and defines groups of related functions into strongly-typed interfaces. Developers can also create custom interfaces.
The basic interface of COM gives a method for components to discover dynamically the interfaces implemented by other components, and it tracks component instantiation by reference counting. COM also includes a mechanism to identify components and interfaces uniquely, and it provides a run-time library to establish and coordinate component interactions. Since component reuse is important, COM supports several black-box reusability mechanisms (e.g., containment, delegation, and aggregation).

SOM

IBM's SOM is the linchpin of its object-enabling infrastructure. SOM is CORBA-compliant, and it will underlay all of IBM's object technology product offerings (e.g., OpenDoc, Taligent frameworks, and the Workplace operating system family). SOM is presently available in the OS/2, AIX, Windows, and Mac System 7 operating systems, and will soon be available in Novell's Netware, as well as IBM's Workplace, MVS, and OS/400.

SOM enables the development of system objects, which can be supplied as part of an operating system, vendor tool, or application. System objects can be distributed and subclasses in binary form, so developers of class libraries need not supply source code. This subclassing can be achieved across languages. Hence, SOM is a packaging technology and run-time support system that allows for the construction of language-independent class libraries.

In addition, the enabling technology allows for subsequent modification of components without recompilation; hence, system libraries need not be rebuilt each time a library component is altered. We note that although SOM is an object-enabling technology, it does not provide compound-document functionality; OpenDoc, layered above SOM, makes those provisions.

SOM is a peer-to-peer communications vehicle interconnecting objects and frameworks with each other, rather than to a central "master" controller. Frameworks are interrelated sets of SOM objects designed to solve a particular problem, and SOM comes bundled with many frameworks.

For example, a distribution framework seamlessly extends SOM's internal method-dispatch mechanism, so that object methods can be invoked transparent to the programmer, on objects in different address spaces or in different machines. Additional components can be added (e.g., marshaling and transport) to support message passing between objects on different machines. Components can also be replaced, depending on the particular distributed computing environment that needs to be supported. SOM achieves cross-language interoperability by building its method-dispatch mechanism based on system-defined procedure-linkage conventions.

SOM and COM define their object models differently; with SOM, a programmer writes code that uses the object infrastructure that SOM provides, whereas with COM, a programmer must also write the code that implements many of the rules that comprise the COM infrastructure. Also, presently COM does not support the cross-machine, distributed-object capabilities of SOM. In addition, Microsoft has bundled COM and OLE to the point that distinguishing between operations of the two is difficult; by comparison, SOM is a complete implementation of a syntax-free, object-oriented, run-time engine engineered to provide a robust binary interface completely encapsulating the implementation details.

SOM's advantage to a developer is that class libraries can be built that support robust binary interfaces. Client programs can be derived from library classes using normal object-oriented inheritance techniques, without having to change internal features of those libraries, and without requiring all client programmers to develop in the same language.

APPLICATION-LEVEL VIRTUAL MACHINES

Application-level virtual machines are at a level above that of interoperable objects and processes; they generally provide a foundation for component object applications. Examples of application-level layers are three compound-document technologies:

- OpenDoc from Component Integration (CI) Laboratories (founded by a group of companies including Apple, IBM, Novell, SunSoft, and Taligent),
- Object Linking and Embedding (OLE) from Microsoft, and
- Taligent Application Environment (TalAE) from Taligent (founded by a group of companies including Apple, IBM, and Hewlett-Packard).

The Cost-Benefit Model: Yardsticks for porting using Application-level machines

When considering a port at the application-level, the costs and benefits of designing using virtual machines can be compared using the following model. These tradeoffs must be carefully weighed to determine the suitability of using a virtual machine for this layer of a given project. The following questions are relevant for application developers when considering compound documents:

- What are the types of data?
- How can new data types be developed?
- What are the operations that can be performed on data?
- How are the interfaces to objects specified?
- How are handles to objects obtained?
- How are client/server mechanisms handled?
- What are the mechanisms for component development and use?
- What are features specific to the object model being considered?

Compound Documents

For the following discussion, we define compound documents to contain many types of data (e.g., text, graphics, tables, video, sound, and animation). The documents can be edited, printed, circulated in read-only format, presented as a slide show, or circulated for mark-up and review. It reduces software complexity by using replaceable binary objects. Application modules interoperate in a seamless fashion by using a single, shared user interface. This frees the user to view work in a document-centric way, as opposed to thinking of work as a set of distinct applications. In addition, it allows users to build documents that can be used on a variety of different platforms, with porting achieved automatically. In this sense, a virtual machine is provided by the very technology that supports compound documents.

Furthermore, through the use of distributed objects, compound documents can reach across networks and transparently participate in client/server arrangements, pulling data from many sources into a single document. Content can be dynamic and automatically generated by a remote database query on-the-fly each time a given document is opened. When this notion is combined with the ability to have multiple representations of information within a single document (e.g., the document can be opened in English or Japanese), we see how this technology really will drive the virtual machine concept at the highest level well into the next century.

OpenDoc

OpenDoc is a system that enables scripting multimedia documents, but it is also truly the basis for an application-level virtual machine. One goal of OpenDoc is to enable application development using small interoperable objects that have well-defined interfaces for representation on computer screens and on paper, and for archival storage. OpenDoc uses a multimedia capable storage format called Bento, developed initially at Apple. OpenDoc uses IBM's SOM for its interoperable object layer. The virtual machine is provided by the interface between objects developed by the user, the SOM layer, the storage format, and the screen/paper presentation.

OpenDoc is an enabling technology that restructures application development in a way that fosters small, reusable, interoperable application components that users can employ in accordance with their needs. It features a unified, but customizable, system for creating complex documents, with a consistent user interface. From the user's perspective, OpenDoc provides a wealth of tools for pasting various content types onto an extensible working environment; new capabilities and content types are furnished in a third-party market of object components known as "parts." Third-party scripting languages also enable specific solutions for honed for specific document content types. And, the flow of events within OpenDoc is made visible to the parts of a document, so that the user's actions can be recorded and played back. These events are not just keystrokes and mouse clicks, but rather semantic actions at a higher level of abstraction, allowing sophisticated creation and modification capabilities.

The methodology for developing an OpenDoc part is straightforward. First, the part's content model and semantic events are defined. Then, the customized, core data engine (e.g., algorithms and data structures specific to the given content type) are implemented, after which the storage-manipulation code is implemented. Next, the part's rendering (e.g., layout negotiation and update support) and user interface event-handling codes are implemented. In addition, the part's scripting code for resolving external references, and desired extension interfaces (if any) are developed. Finally, the part is packaged with documentation and made available for insertion by other users.

OLE

Like OpenDoc, Microsoft's OLE [Brockschmidt 95] is a unified environment of object-based services with the capability of both customizing those services and arbitrarily extending the architecture through custom services, with the overall purpose of enabling a rich integration between components. Its key customizable services are globally usable by all applications, by the system, and by all other services. OLE allows users to add features and technologies to its framework, without changing the framework.

In OLE, a client is a user of objects, and a server is a furnisher of objects. A service consists of one or more binary components, each consisting of one or more objects. Objects provide functionality and content through one or more interfaces. The concepts that form the idea of an OLE object are collectively called COM, described in the interoperable objects virtual machine layer. Objects feature three practical properties:

- *Encapsulation.* The details of composition, structure, and internal workings of an object are hidden from the clients, which see only the interface.

- *Polymorphism.* Simply put, polymorphism provides the ability to view two similar objects through a common interface.

- *Inheritance.* Base classes of objects can lend properties and characteristics to derived classes, and this inheritance can be employed hierarchically, permitting specialized designs.

OLE allows the integration between binary components made up of Win32-style Applications Programming Interface (API) functions. In its model, a software component is a reusable piece of code and data in binary format that can be plugged into other components with minimal effort. OLE allows users to purchase binary components and reuse them, plugging them in through external interfaces.

OLE is about integration on many levels: Components can come in various forms, such as simple functional objects with straightforward interfaces (e.g., a string object), automation objects, data sources, compound-document objects, or controls. The many services offered by the integrated OLE engine include structured storage, object persistence, name persistence, moniker typing, uniform data transfer and drag-and-drop, notification, and automation. Although no support currently exists for distributing objects, future incarnations of OLE will make provisions for such capability.

TalAE

Taligent's system architecture provides a fully object-oriented programming environment and a powerful set of software building blocks. The major aspects of system software (e.g., development, operation, and applications) are included in Taligent's web of services as reusable sets of software called "frameworks." Frameworks are collections of objects that provide an integrated service intended for customization by the developer; they are small, manageable sets of functionality at a level of abstraction set between a large class library and a single class. Frameworks provide both infrastructure and flexible interfaces, and are designed to be customized to create solutions for families of related problems.

Taligent's pool of frameworks is divided into three main areas: Taligent Application Environment (TalAE), which includes most of the object-oriented frameworks; Taligent Development Environment (TalDE), which is an extensible set of tools frameworks designed expressly for object programming; and Taligent Object Services (TalOS), which provides extensible frameworks for more traditional operating system functions. The open structure of Taligent's frameworks enables third parties to extend and customize the system at all levels; in addition, Taligent supports the CORBA specification for distributed-object access in open, multisystem environments.

TalAE is distributed and portable, and contains over 100 frameworks, spanning such areas as graphics, database access, multimedia, user interface, internationalization, networking, and distributed computing. TalAE runs on existing 32-bit operating systems (e.g., OS/2, AIX, HP-UX, and PowerOpen). TalAE is complemented by TalDE, a suite of framework-based developer tools. TalDE provides capabilities for developing dynamic browsers, incremental automatic building with incremental development, online documenting, multiuser source code controlling, and GUI constructing. TalOS is an object-oriented operating environment for hosting TalAE, and it uses a Mach-based microkernel enhanced for objects.

Taligent frameworks are integrated in a coherent architecture that allows them to function with full interoperability. Since TalAE uses frameworks at all levels, the distinction between the system and its applications is blurred, allowing for full object-oriented development of applications on a virtual machine level.

Summary

OpenDoc and similar document-centric virtual machines simplify the portability of applications, especially as new display devices are discovered. Porting the application only requires porting the OpenDoc layer. Similar advantages hold for TalAE, OLE, and other application level virtual machines.

CONCURRENT VIRTUAL MACHINE: A SUMMARY

1. *A few virtual machines.*

   It is unlikely that a single virtual machine will be appropriate for all DOD applications. Virtual machines at different levels will be required to help maintain applications as they are ported from one kind of machine, operating system, programming language, interoperable object layer, and application layer to another. The virtual machine at one layer is not always appropriate to other layers.

2. *Consistency across virtual machines.*

   It is helpful to have layers of virtual machines that have design consistency at all levels from the hardware up through the application. For instance, one consistent view is to have a network of multiprocessor workstations at the hardware layer, process/thread virtual machines at the operating systems layer, an object-oriented language with support for distributed interoperable objects such as C++ or Smalltalk with CORBA, and an application layer such as OpenDoc. This consistency across virtual machines is based on the interoperable object as a central idea. Other philosophically consistent layers of virtual machines can be constructed.

3. *Standard virtual machines.*

   One of the goals of using virtual machines is to benefit from investment in applications over long periods of times as machines and application specifications change. A key issue in this approach is the effort required to map (or translate or compile) a virtual machine at one layer to one at another layer. Commercial off the shelf virtual machines (e.g., CORBA implementations such as DSOM) are so widely used that the development of mapping technologies to new platforms will be less of a problem than mapping proprietary virtual machines that are not widely used. Therefore, we should pay particular attention to COTS technologies to determine what technologies can be used as virtual machine layers.

4. *Exceptions.*

   There may be a few applications that are developed without using standard virtual machines. Porting and maintaining these applications will be difficult, but in rare cases, use of nonstandard machines, operating systems, programming languages and interoperable object or message layers may be appropriate. The intent of using virtual machines is to retain investment in most (but not necessarily all) applications, over time.

5. *Threads processes objects and communication layers.*

A virtual machine that spans all layers is one that deals with threads, processes or objects, and interoperable objects or communication layers. Several programming languages (including Ada) and object-broker based systems are consistent with these issues. We should pay particular attention to virtual machines based on these concepts, and explore their utility for scientific computing and other applications.

APPENDIX 3: CASE STUDY --- TERRAIN MASKING

INTRODUCTION

The goal of this section is to evaluate application specific concurrent virtual machines. An application-specific concurrent virtual machine is also called a concurrent program archetype. The question that we wish to evaluate is whether the development of a concurrent program archetype is helpful in reducing the amount of programming effort to produce efficient parallel implementations of problems that fit the archetype. The example problem that we picked is the problem suggested by Rome Laboratory:

Terrain Masking.

In this report we describe the problem, design concurrent programs using different programming paradigms, and finally discuss a problem archetype obtained by generalizing this problem. Program outlines for different concurrent programming languages are provided.

We apply the Virtual machine based methodology proposed in this document to the design of a program for terrain masking. This benchmark is one of a suite of benchmarks being developed by Honeywell Corporation called the C3IPBS (C3I Parallel Benchmark Suite) for the purpose of evaluating the suitability of parallel hardware platforms and software development tools and processes for C3I applications. We present virtual machines at each of the levels introduced in the previous chapters, present a refinement methodology using which a virtual machine can be refined into a less abstract virtual machine, and present a cost-benefit evaluation of the virtual machine at each level of abstraction. We conclude the chapter with an evaluation of the virtual machine based methodology for the class of scientific applications represented by this benchmark.

Problem Statement

Terrain masking computations are typically used to identify the regions in space that are not visible, or are masked from, a set of threats with known range and positions. Such computations are used either to plot aircraft flight plans with low probabilities of detection or to position threats to maximize the area of coverage. The basic problem is demonstrated in Figure 4: given a threat's location and the attributes of the surrounding terrain, 'line of sight' calculations are used to identify regions of space within a predefined *gaming area*, that are invisible and hence masked from the given threat. The preceding terrain masking calculation is performed for each threat, and the results are merged over all threats using an appropriate combining operation to compute the terrain masking for a given threat scenario.

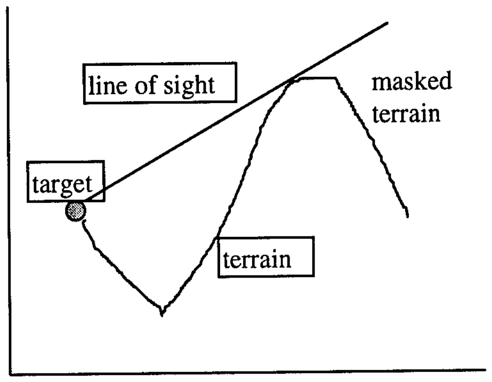


Figure 4: Terrain Masking Problem

Problem Specification

A virtual machine for this application at the level of problem specifications is as follows: let

- *T* and *t* respectively refer to the set of threats and an arbitrary threat in the set.
- *X* and *x* respectively refer to the set of points in the gaming area at which the terrain masking is to be computed, and to an arbitrary point in the set in*X*.
- x_t refers to location of threat *t*,

Let *v(x,t)* represent the impact of threat *t* at point *x*. We are required to compute the impact of the threat at each *x* in the set *X* . In this benchmark, we assume that the impact at a point *x* is represented by the *masking altitude*, which is the maximum altitude at*x*, such that the terrain below this altitude is masked at point x. For a given threat*t*, *v(x,t)* is computed as follows:

- $v[x, t]$ is given for $x=x_t$
- $v[x,t] = f(v[x_1, t])$, for $x \neq x_t$, where x_1 is the closest point to *x* that lies in the set *X* and lies on the line of sight fromx_t to *x*.

The specification requires that starting at the location of the threat, the impact of the threat at different points be computed in a direction stretching away from the threat itself. Let w[x] represent the cumulative impact of all threats at point x. In this benchmark, the cumulative impact is determined by a function*g*. This cumulative impact for a given point *x* is specified by

- $w[x] = g\ (v[x,t_0],\ v[x,t_1],\ v[x,t_2],\ \ldots\ v[x,t_n])$ *for each x in X*

The preceding virtual machine at the specification level is perhaps the simplest possible VM for this class of problems. Changes in the specification of the problem are easy to incorporate and the program is easy to understand. Of course, direct implementations of this VM on most existing architectures will tend to be inefficient.

Parallel Implementations

Uniform Memory Access VM

The preceding specification can be refined to a UMA CVM as follows:

```
In parallel for all t in T do
   sequentially for all x in X do   compute v[x,t];
In parallel for all x in X do   compute w[x];
```

Figure 5: Specification CVM for Terrain Masking

The program in Figure 5 can be refined to allow the *v[x,t]* to be computed in parallel for some points. For instance, points that lie within concentric regions stretching away from a threat may be processed in parallel as illustrated inFigure 6. In the figure, assume that *v[p,x]* has been computed for all points that lie within the darker shaded area surrounding the threat. Points that lie within the lightly shaded region surrounding this may be processed in parallel to compute the impact due to the threat. A number of programming languages use the uniform memory access programming model to specify parallel programs. One such language, UC, was described in a previous chapter. We present UC programs for the terrain masking problem. Figure 7 presents the implementation specified by the UMA CVM program of Figure 5 and Figure 8 contains the version with the additional parallelism indicated by theFigure 6.

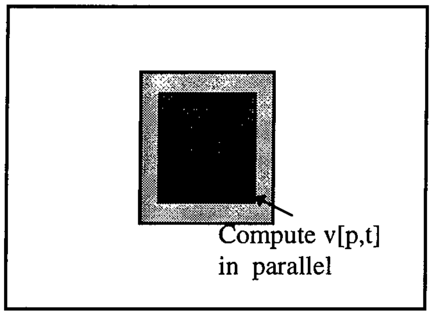


Figure 6: Additional Concurrency

```
index set T:t = 1..N   /*  assume there are N threats */
index-set X:x = 1..P  /*  assume here are P points in set X */
int masking-altitude[P,N], minimum-masking-altitude[P];
par (X,T)
        masking-altitude[x,t] = appropriate-initial-value;

par (T)     /* In parallel, for all threats in the gaming area*/
  for ($x=x_t$; in-range(x,t) ; x=inc(x) )  /* sequentially traverse pts in x that
                                               are in the range of threat t, starting
                                               from the location of the threat */
        masking-altitude[x,t] = compute-masking-altitude(x,t,v);

par (X)
  minimum-masking-altitude[x]  = $< { T; v[x,t]};  /*  compute w[x] for
        all points */
```

Figure 7: Data Parallel UMA CVM

```
index set T:t = 1..N   /*  assume there are N threats */
index-set S:s = 1..NS /*  assume there are NS slices*/
index-set X:x = 1..NP_S  /*  assume there are P points in  each slice*/
int masking-altitude[P,N], minimum-masking-altitude[P];
par (X,T)
        masking-altitude[x,t] = appropriate-initial-value;

par (T)     /* In parallel, for all threats in the gaming area*/
  seq (S)  /*  Sequentially traverse each slice*/
     par (X) st (in-range(x,t)   /*  In parallel for all  pts in the slice that are
                                     in the range of threat t   */
           masking-altitude[x,t] = compute-masking-altitude(x,t,v);

par (X)
  minimum-masking-altitude[x]  = $< { T; v[x,t]};  /*  compute w[x] for
        all points */
```

Figure 8: Data parallel UMA CVM with Additonal Concurrency

Parallel Implementations -- SPMD CVM

In a NUMA (Non Uniforma Memory Access) VM, the set of threats must be explicitly divided and allocated among the individual processors and memories. For a homogenous threat scenario, a load balanced decomposition can be easily achieved simply by dividing the threats among the available nodes. For heterogenous threats with widely different ranges and/or nodes with different capabilities, the threat partitioning must try to allocate threats to nodes such that the computational load is approximately balanced among the processors. Performance may also be improved by grouping threats that are in the proximity of each other on one processor. An alternative to static threat partitioning is to use dynamic partitioning, where threats are managed by one (or a few) managers. Each node is initially allocated a small number of threats; subsequently each node requests threats from the manager(s) when it has finished processing its assigned quantum.

We develop a SPMD program that uses static partitioning of the threats (Figure 10) The program is a simple refinement of the UMA (Uniform Memory Access) CVM described in the previous section. We present the program using a pseudo-code notation. Example languages discussed in the previous section that support this computation model include C or FORTRAN with MPI. The primary distinction between the program in this section and the one develped in the previous section is the use of explicit barriers to synchronize the execution of the program on the multiple nodes. In the version developed in this section, we assume that each node of the parallel architecture is a single processor. We subsequently refine this program where each node may itself be a multiprocessor.

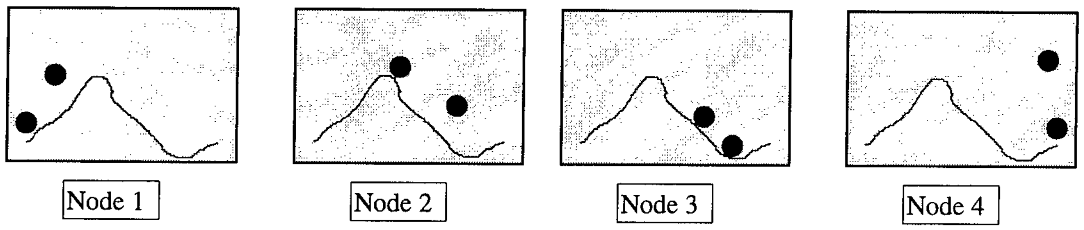


Figure 9: Threat Decomposition

The pseudo-code program for this decomposition is given inFigure 10.

```
program main;
{ index set N:n = 1..C   /* assume there are C processors*/
  index set T:t = 1..N    /* assume there are N threats */
  index-set X:x = 1..P    /* assume there are P points in set X */
  int masking-altitude[P,N], minimum-masking-altitude[P];

   if ( MPI_comm_rank(...,myid, ...)==0)
    { read threat-data-file;
       for (i=0; i<C; i++)           /* assume there are C processors */
         partition[i] = compute-threat-partition(T);
    }
   MPI_scatter( ...,partition, ...) /* each processor is sent the subset of its threats */
   for all (X,T) st ( is-in-my-partition(t))
        masking-altitude[x,t] = appropriate-initial-value;

   for all (T) st ( is-in-my-partition(t)) /* for all threats in 'my' partition */
      for (x=x_t ; x=inc(x); in-range(x,t) ) /* traverse pts in x that are in the range of threat
                                                     t, starting from the location of the threat */
        masking-altitude[x,t] = compute-masking-altitude(x,t,v);
   /* compute minimum altitude for threats in local partition */
   for all (X)
      local-min-alt[x] = min : ( is-in-my-partition(t)) : masking-altitude[x,t]

  /* compute minimum altitude across all partitions */
  MPI_REDUCE(local-min-alt, minimum-masking-altitude, ...);
}
```

Figure 10: SPMD CVM for Terrain Masking with Threat Partitioning

An alternate SPMD design is to partition the gaming area among the available processors and replicate the set of threats. Once again the gaming area must be partitioned into uniform or non-uniform segments based on the nature and distribution of threats so as to approximately balance the computational load. Similar to the case of threat partitioning, it is possible to use dynamic partitioing of the gaming area. The decomposition of the problem is shown inFigure 11. A primary advantage of the decomposition of the gaming area rather than the threats is that the minimum masking altitudes for each grid point in the gaming area does not have to be communicated among the nodesas it did in the case where the problem was decomposed by dividing the threats among the available processors. However, as the masking altitude at each point due to a given threat is computed in an outward direction starting from the location of the threat, in general, each processor must send the masking altitude computed along its boundary to the neighboring processor. In Figure 11, node 3 must wait until node 1 has completed the computation of the masking altitude for threat *t1* for points along its south boundary and has sent the needed information to node 3. Of course, in the meantime, node 3 can compute the masking altitude for threats *t2* and *t3*. However in threat scenarios where the threat distribution is uneven across the gaming area with many of the threats concentrated in one region, this decomposition could yield a fairly unbalanced computational load among the processors. The program corresponding to this decomposition is presented in Figure 12.

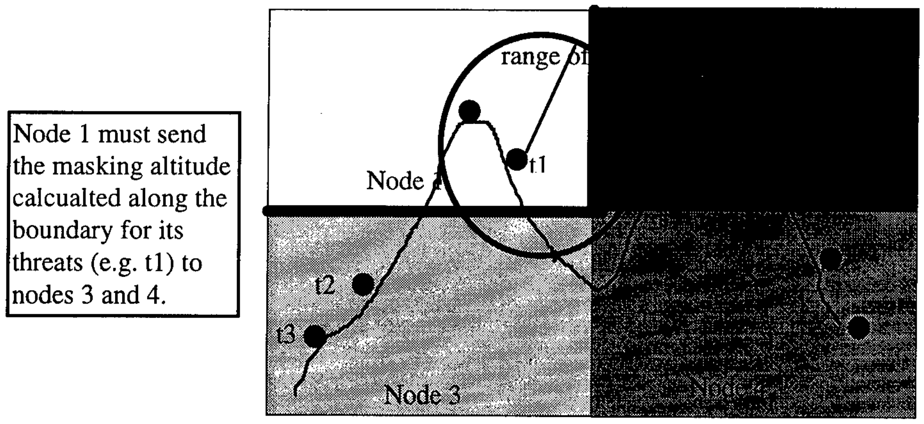

Node 1 must send the masking altitude calcualted along the boundary for its threats (e.g. t1) to nodes 3 and 4.
range of
Node 1
t1
t2
t3
Node 3
Node 4

Figure 11: Decomposition Of Gaming area

```
program main;
{  index set N:n  = 1..C   /*  assume there are  C processors*/
   index set T:t = 1..N    /*  assume there are N threats */
   index-set X:x = 1..P    /*  assume there are P points in set X */
   int masking-altitude[P,N], minimum-masking-altitude[P];

   read threat-data and terrain-data files for corresponding  partition;
   for all (X,T)  st ( is-in-my-partition(t))
         masking-altitude[x,t] = appropriate-initial-value;

   for all (T) st ( is-in-my-partition(t))  /* for all threats in  'my' partition */
   {    for (x=$x_t$; in-range(x,t) && in-partition(x) ; x=inc(x) )
             /* traverse pts in X within the partition  and in the range of threat  t */
             masking-altitude[x,t] = compute-masking-altitude(x,t,v);
         MPI_SEND masking-altitude at boundary to neighbor partitions with tag DATA;
   }

  /* After computing masking-altitude for threats in its partition, wait to receive
      boundary element values from neighbor partitions   */
   DONE=0;
   while ( !DONE)
   {   MPI_RECV (....,ANY_SOURCE, ..,ANY_TAG,...status, ...);
       /*  The receive updates a variable calledstatus which  includes info on theTYPE
           and SOURCE of the incoming msg  */
         if (status.TAG == DATA)  /* msg contains masking altitude for a threatt  */
            for all  (X) st (in-partition(x))
               compute masking-altitude[x,t] for corresponding threat t;
```

```
        DONE= all-neighbor-msgs-recd(recv-set, status);   /* check if msgs recd from all
                                                             neighbors for all threats    */
    }
   /* compute minimum altitude for all points in X that lie in local partition */
    for all (X) st (in-partition(x))
      minimum-masking-altitude[x] = min : T : masking-altitude[x,t]

}
```

Figure 12: SPMD CVM -- Partitioned gaming area

Adding Threads

If each node of the parallel architecture is either a multiprocessor or can support threads, the amount of concrrency can possibly be increased, by further decomposing the threats or gaming area assigned to each partition. As seen previously inFigure 11, a node must typically wait until a neighbor node has finished computing the masking altitude along the entire boundary that is shared between the two. However, if the the area assigned to each node is further decomposed into smaller partitions, each of which is assigned to a thread, then each thread need only wait for the boundary that the thread's partition shares with the neighbor, which is typically smallerFigure 13 shows the decomposition and communication boundaries in the presence of threads:

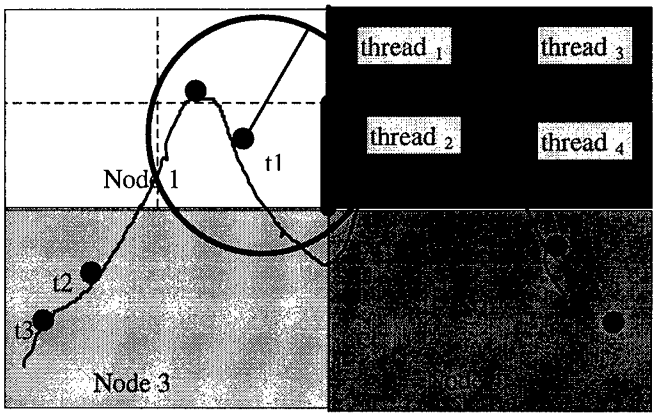


Figure 13: Thread-based CVM for Terain Masking

Given the thread based decomposition demonstrated inFigure 13, the computation in node 1 is performed using four threads. The thread that includes threat *t1* can compute the boundary elements at the boundary with node 2 and send the boundary elements to thread$_2$ in node 2. This can increase the amount of available concurrency in the system because now thread$_2$ need not wait until the computation corresponding to the entire region assigned to node 1 is completed; instead each thread simply waits for the boundary values that correspond to its region and can immediately begin to compute thereafter. Of course, the threads that execute together on a node also need to communicate the boundary elements to each other. However, if the threads on a node execute in a common address space this communication can typically be very efficient as it may be implemented using pointers to a common address space. Such an integrated model is developed in a subsequent section.

Task parallel CVM

The preceding SPMD CVM can easily be refined into a task parallel CVM for the terrain-masking benchmark. The primary difference between a SPMD CVM and a task parallel or Multiple Program Multiple Data (MPMD) CVM is that whereas the former requires that exactly the same program executes on all processors, a MPMD program may have different programs (or processes) executing on different processors. Programs with dynamic process creation and termination typically belong to the MPMD class. However as in the case of the terrain masking benchmark, there is no need for dynamic process creation, all nodes execute exactly the same program so the task parallel CVM is identical to that of the SPMD CVM presented in the previous section.

Integrated task and data parallel CVM

The most common form of an integrated task and data parallel computation is a model wherein different data parallel program segments can execute in parallel with each other on different nodes of a parallel architecture. In most of the CVMs presented in the preceding sections, the computation of the *minimum-masking-altitude* at any point in the gaming area can proceed only after the *masking-altitude* has been computed *at all points* for all threats. However, as clearly indicated by the specification CVM, the *minimum-masking-altitude* at a point *x* is determined only by the *masking-altitude* due to all threats at that point. As a result it is possible to increase the amount of concurrency in the program by interleaving the computation of the *minimum-masking-altitude* in an area where the I*masking-altitude* has already been computed with the computation of the *masking-altitude* in the remainder of the gaming area as demonstrated by the overlapping in Figure 14. Assuming that the *masking-altitude* has been computed *for all threats* within the darker inner shaded area, the minimum-masking-altitude can be calculated within the inner area while the masking-altitude is being computed in the outer region.

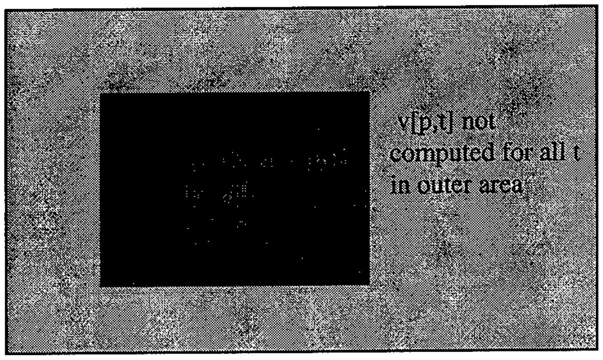


Figure 14: Overlapping computations

Program Archetypes or Templates: Problem-Specific Virtual Machines Using Concurrent Objects

This section explores the use of concurrent objects to define program archetypes or templates. The central idea is to: (1) define an abstraction common to a class of related problems, (2) develop a program archetype (or skeleton or template) common to all problems in the class, and (3) develop parallel programs for a given problem in the class by specializing the archetype. Thus the central idea is to develop a hierarchy of virtual machines where the most abstract virtual machine in the hierarchy is specific to a class of problems and is applicable to all architectures. More refined virtual machines are tailored to a specific class of target architectures such as shared-memory multiprocessors or

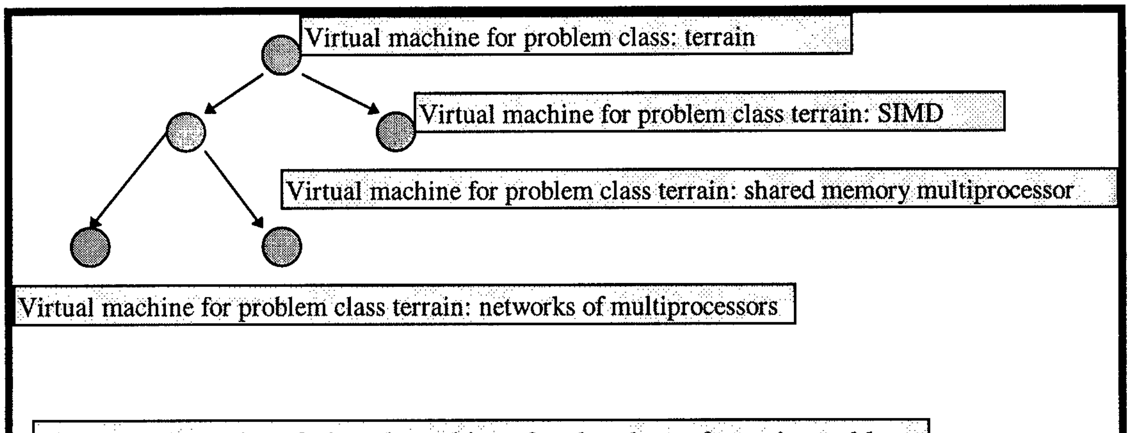

Virtual machine for problem class: terrain
Virtual machine for problem class terrain: SIMD
Virtual machine for problem class terrain: shared memory multiprocessor
Virtual machine for problem class terrain: networks of multiprocessors

Figure: Hierarchy of virtual machines for the class of terrain problems

message-passing networks of multiprocessors (that must consider both 1. processes that communicate with each other using messages, and 2. threads within processes where all threads within a process share memory). An efficient program for a specific architecture for a given problem within the class can be obtained by specialising an archetypal program for the specified architecture. For instance, an efficient program for the terrain masking problem specified by Honeywell, that executes on a shared-memory multiprocessor can be obtained by specialising the archetypal program for this class of problems designed for efficiency on shared-memory multiprocessors; such an archetypal program is described later.

The Archetypal Problem

Inputs:

1. A rectangular grid with dimensions*xlength*, *ylength* where *xlength* and *ylength* are integers.
2. A two dimensional array terrain of dimensions*xlength*, *ylength*, where for each point *(x,y)* on the grid, *terrain[x,y]* is of some type *terrainType.*
3. A nonnegative integer: *numThreats.*
4. A one-dimensional array*threatInformation* where *threatInformation[t]* is of type *threatInformationType*, where this type includes the position of threat*t* on the grid, the range of *t*, and other information about threat *t*.

Intermediate Computations

1. *threatEffect*: three-dimensional array of dimensions *xlength*, *ylength*, *numThreats* where *threatEffect[x,y,t]* is the effect of threat *t* at point *(x,y)* on the grid. The value of *threatEffect[x,y,t]* is significant if and only if *outOfRange[x,y,t]* --- see next paragraph --- is *false*.

2. *outOfRange*: three-dimensional boolean array of dimensions *xlength*, *ylength*, *numThreats* where *outOfRange[x,y,t]* is true if and only if point *(x,y)* on the grid is out of range of threat *t*. Associated with each threat is a range, and points that are of distance greater than the range from the threat location are not effected by the threat.

3. *overallEffect*: two dimensional array of dimensions *xlength*, *ylength* where *overallEffect[x,y]* is the total effect of all the threats *t*; it is a function of *threatEffect[x,y,t]* for all *t*.

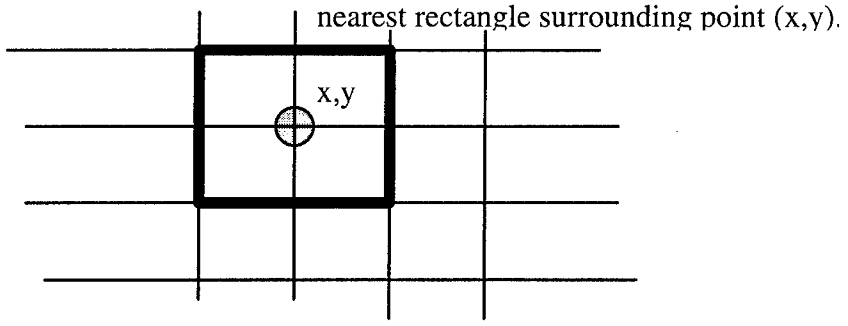
Data Dependencies in the Problem Archetype

Computation of *threatEffect*.

Consider a threat *t*, and let *txLoc* and *tyLoc* be the *x*, *y* coordinates of the location of threat *t*. The values of *txLoc* and *tyLoc* are given in *threatInformation[t]*.

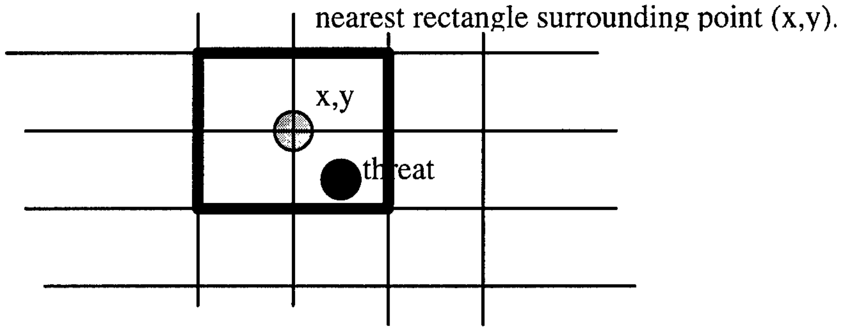
The value of *threatEffect[txLoc, tyLoc, t]*, i.e., the effect of a threat *t* at the point at which the threat is located is specified by a function, *t.threatEffectAtThreatLocation( )*.

The value of *threatEffect[x,y, t]*, is computed as follows. Consider the straight line segment connecting point *(x,y)* on the grid and the location *(txLoc, tyLoc)* of threat *t*. We distinguish three cases depending on the relative location of the point, the threat and the rectangle consisting of the grid lines nearest the point that do not pass through the point; this rectangle is shown in the figure below:

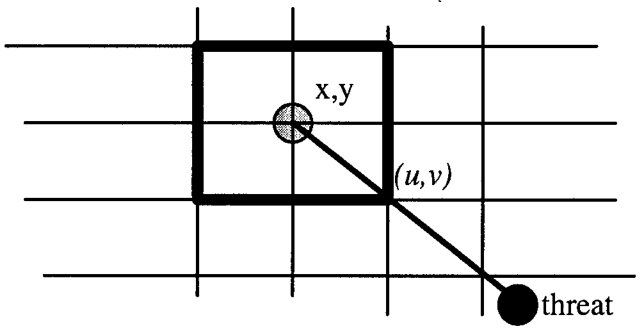


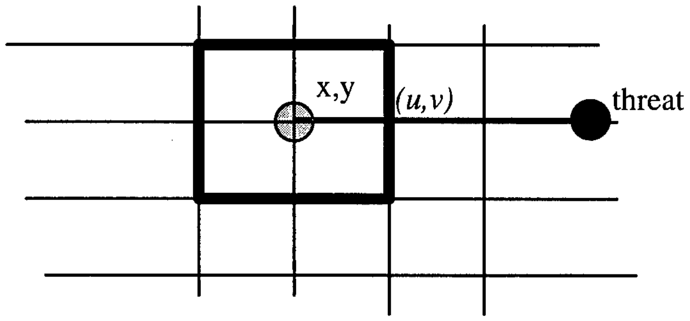
The three cases are as follows:

- The threat location and the point are both within the rectangle.

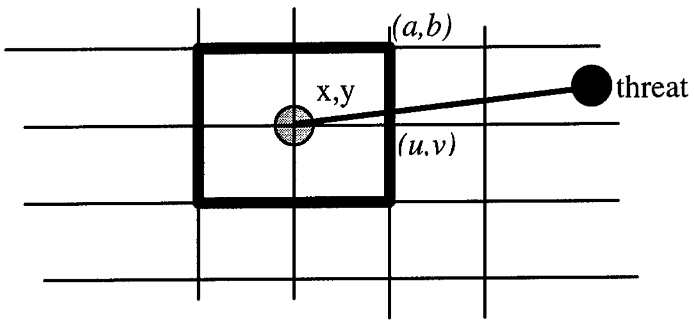


- The straight line passes through one of the 8 grid points on the rectangle:





- or the straight line intersects an edge of the rectangle, but not at a grid point.



The value of *threatEffect[x,y]* is computed for each of the three cases in the following way:

1. In the first case, a function *atThreatLocation* is used,

   threatEffect[x,y, t] = atThreatLocation(x,y,t)

2. In the second case, a function *lineThroughGridPoint* is used to compute *threatEffect[x,y, t]* as a function of *threatEffect[u,v, t]* where *(u,v)* is the grid point through which the line connecting*(x,y)* to the threat location passes.

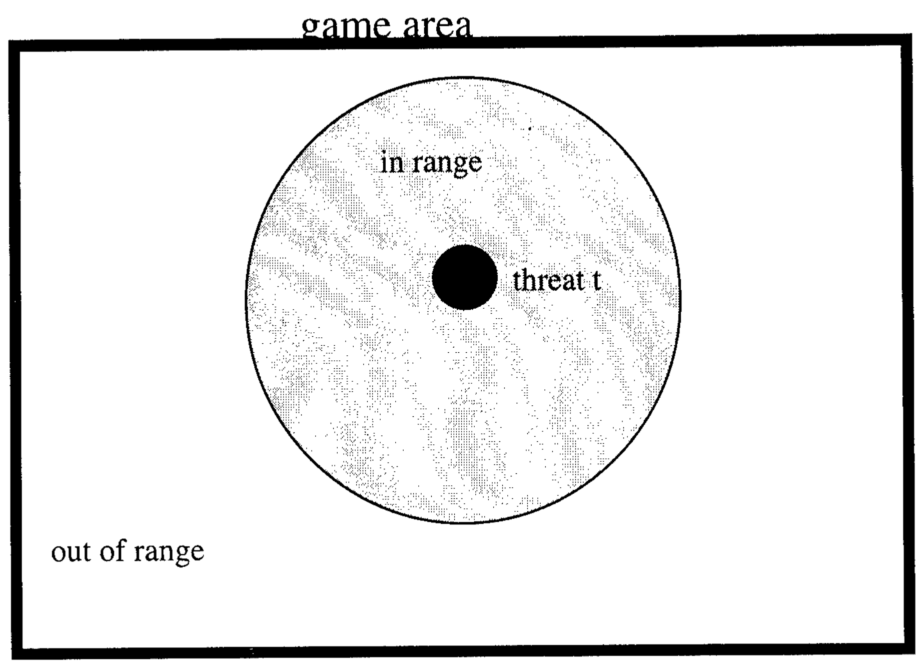
   threatEffect[x,y, t] = lineThroughGridPoint(x,y,u,v,t)

3. In the third case, a function *generalCase* is used to compute *threatEffect[x,y, t]* as a function of *threatEffect[u,v, t]* and *threatEffect[a,b, t]* where *(u,v)* and *(a,b)* are the grid points on the rectangle on either side of the point at which the line (connecting *(x,y)* to the threat location) intersects the rectangle. Of course, either $u = a$ or $v = b$, and u differs from a by at most 1, and likewise, v differs from b by at most 1.

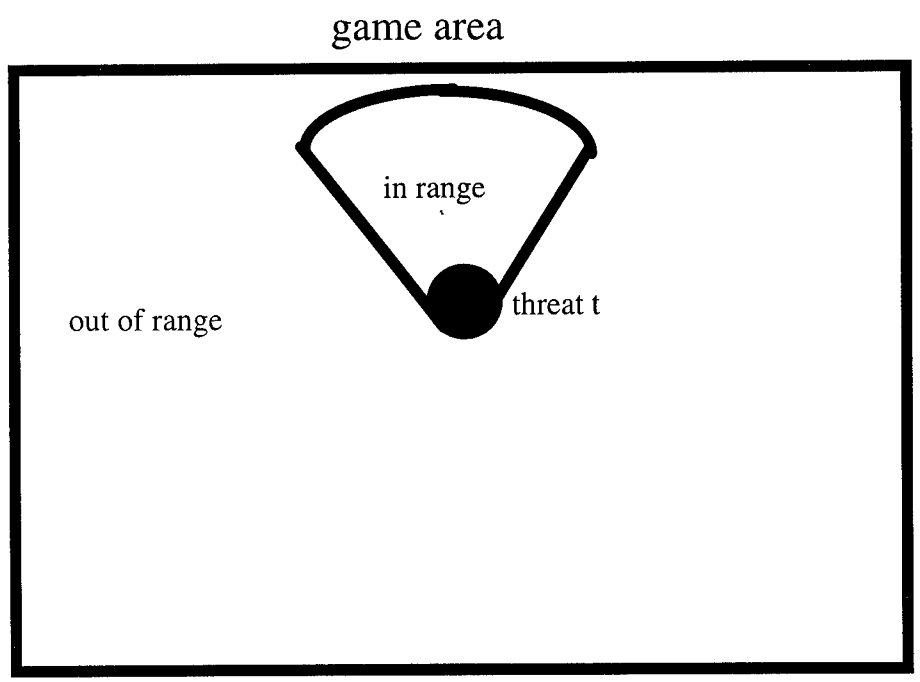
   threatEffect[x,y, t] = generalCase(x,y,u,v,a,b,t)

Computation of *outOfRange*

The meaning of *outOfRange* is: *outOfRange[x,y,t]* is *true* if and only if point*(x,y)* is out



undirectional threat



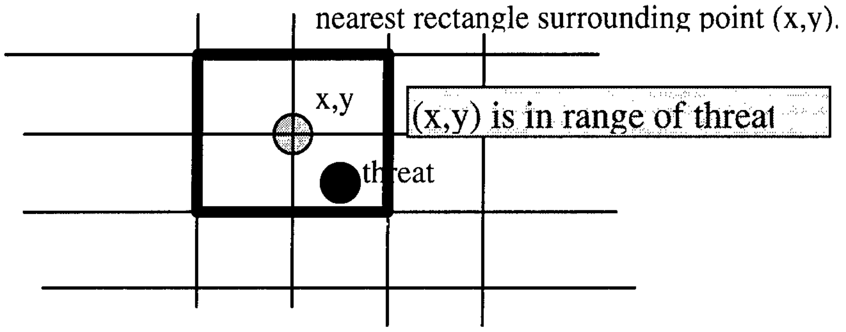
directional threat

of range of threat *t*. The threats can have direction. So, the areas that are within range of the threat can have arbitrary shape.

The value of *threatEffect[x,y,t]* is immaterial if point*(x,y)* is out of range of threat *t*, i.e., the computation does not use *threatEffect[x,y,t]* if *outOfRange[x,y,t]* is true.

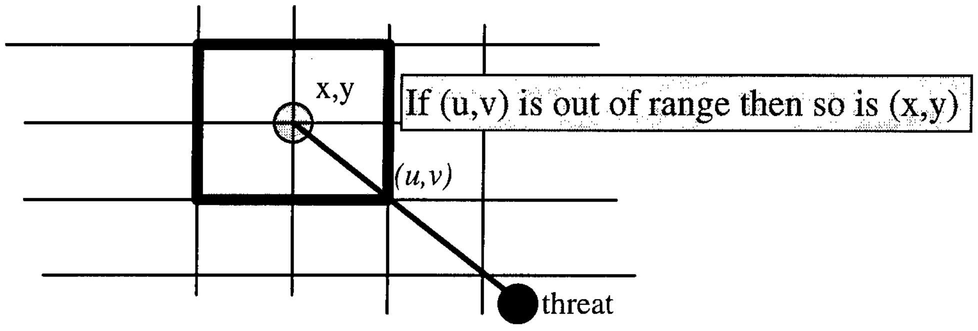
Restriction: We restrict attention to the following kinds of threat ranges. Consider the three cases

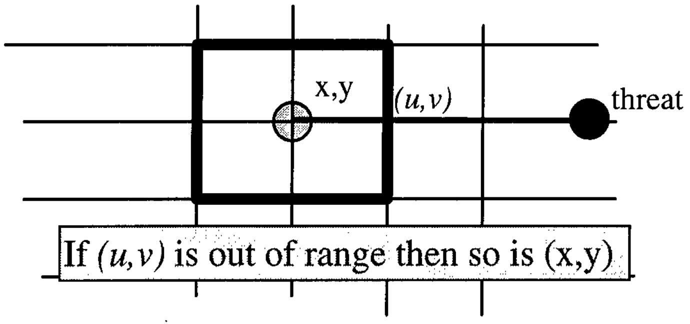
- The threat location and the point are both within the rectangle.


nearest rectangle surrounding point (x,y).
x,y
(x,y) is in range of threat
threat

In this case we assume that point *(x,y)* is in range of the threat.

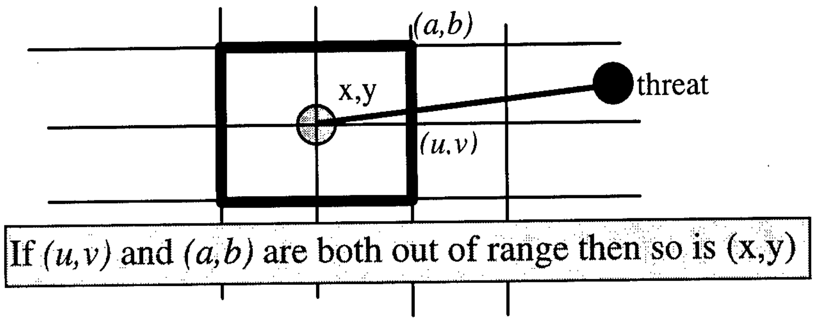
- The straight line passes through one of the 8 grid points on the rectangle:





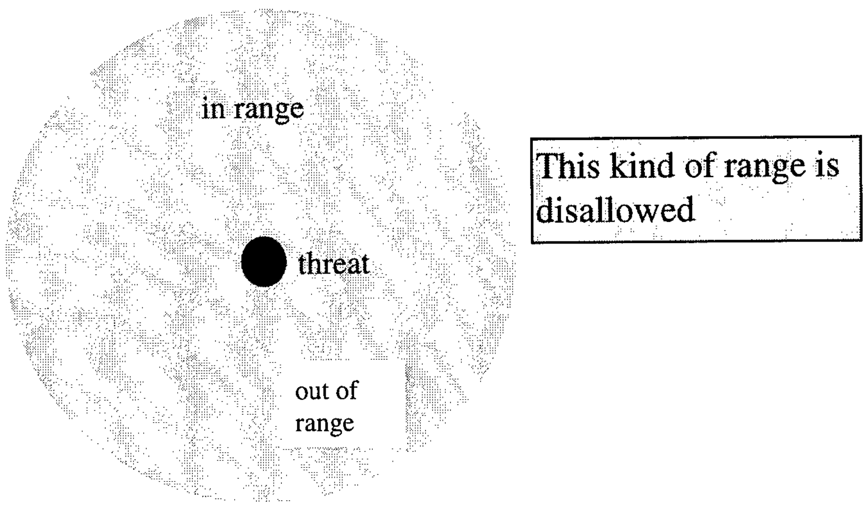
In this case, point *(x,y)* is out of range of the threat if *(u,v)* is out of range of the threat. Note that *(x,y)* can be out of range of the threat even if *(u,v)* is in range of the threat.

- or the straight line intersects an edge of the rectangle, but not at a grid point.



In this case, we assume that (x,y) is out of range of the threat if both (a,b) and (u,v) are out of range of the threat. Note that (x,y) may be out of range even if both (a,b) and (u,v) are in range. Also, (x,y) can be in range if either (a,b) or (u,v) or both are in range.

This restriction disallows threat ranges such as those depicted in the following figure. This restriction is not severe.



<u>Computation of *overallEffect*</u>

The value of *overallEffect[x,y]* is a function of *threatEffect[x,y,t]* for all threats *t* such that point *(x,y)* is in range of the threat. There is an associative commutative operator *op* such that:

overallEffect[x,y] = (*op* over all values of *threatEffect[x,y,t]* such that *outOfRange[x,y,t]* = *false*)

For example, if there are three threats t_0, t_1, t_2, and

outOfRange[x,y, t_0*]* = true, *outOfRange[x,y,* t_1*]* = false, *outOfRange[x,y,* t_2*]* = true

then *overallEffect[x,y]* = *threatEffect[x,y,* t_0*]* *op* *threatEffect[x,y,* t_2*]*

If the effect of a threat is the minimum safe altitude, as in a line-of-sight calculation, then the operator *op* is the min operator.

Templates: Pros and Cons

The central value of templates is reuse. The primary cost-benefit tradeoff deals with the cost of developing a template versus the benefits obtained from reusing the template for several applications. This paragraph is a cost-benefit analysis of templates; this analysis leads us to the subject of the report for this milestone --- the design of a template for an example application. A template is the same as an application-based virtual machine. The virtual machine for a class of applications is, in effect, a template for solutions to that class.

Templates, patterns of objects, program archetypes, and other structures for capturing commonalities among applications are designed to reduce the effort required to develop reliable efficient systems by (a) investing a lot of careful effort in identifying important classes of similar problems, and then developing reliable efficient parameterized template applications for these problems, and (b) developing specific applications within the class by tailoring the template application. The cost-benefit tradeoff is: Does the extra effort in identifying problem classes and template applications pay off later in developing specific applications? This tradeoff depends primarily on the following four questions:

- How general is the problem class? The more times the template application is reused for different problems the more cost effective is the effort spent in developing templates.
- How important is the problem class? Careful design effort expended in making a template reliable and efficient is worthwhile if the class of applications is important.
- How well parameterized is the template? Templates that are carefully parameterized can be tailored for specific applications with less effort.
- How well constructed is the user-interface for the template, and how well-designed are the documentation and help facilities that aid a user in tailoring a general template to a specific application?

Templates for parallel applications in C3I

Template libraries for parallel applications within C4I systems have characteristics commonto other template libraries. Of special concern to parallel applications of C4I are the following issues:

- **Parameterization of the template application so that it can execute efficiently on different architectures**. For instance, can the terrain masking application be parameterized so that efficient programs for networks of symmetric multiprocessors and distributed memory machines be obtained by parameterizing the same template?
- **Parameterization of templates so that users can be shielded from having to deal with details of parallelism**. For instance, can the terrain masking application be parameterized so that the parameters supplied by the user can be defined in familiar sequential programming issues? Or do template parameters deal with parallel issues such as threads, memory coherence, or messages?
- **Composition of applications into C4I systems**. Most applications are used within larger applications. Therefore, a critical question is the ease with which templatized applications can be composed to form larger applications. For instance, how simple is it to compose a terrain masking template within a larger C4I application dealing with optimal location of radar sites on a terrain?
- **Support for mapping programs derived from templates on to target machines**. Does the template library and related tools provide support for mapping programs so as to obtain efficient implementations?

Current Milestone: Evaluation of a parallel template for C4I

This milestone deals with an evaluation of an example template of parallel C4I applications; this template is from a suite developed for Rome Laboratory by Honeywell Inc, and the template application is terrain masking. The goal of this milestone is to design a template including its user interface and supporting tools, and then evaluate the design. As pointed out earlier, an important step is to investigate potential uses of the application within larger C4I systems.

Uses of the application

The terrain masking application is used within larger C4I applications. We consider some applications within which TM (terrain masking) may be used; this will help in setting parameters for the TM template. These larger applications were sugested in meetings with Rome Laboratory, or seem to the authors of this report to be reasonable extensions of TM. The intent here is to consider two or three sample problems within which TM can be embedded to gain an understanding of TM parameters; our intent here is not to have an exhaustive description of applications that use TM.

- **Airborne or underwater sensors**. The TM problem, as specified, has ground based threats, and the problem is to find, for each point on the terrain, the altitude below which an aircraft could fly without being detected. With airborne sensors, such as those on AWACS planes the problem becomes more complex. Likewise, the problem becomes more complex with underwater sensors, and is even more complex with combinations of different types of sensors.

- **Mobile sensors**. The TM problem, as specified, has a set of sensors that are fixed to the terrain. Mobile sensors make the problem more complex. A more difficult issue than the one faced in the static TM problem is to determine "safe trajectories" for aircraft given the patrol paths of mobile sensors.
- **Optimal placement of sensors**. In the given TM problem, the location of sensors is given. A more complex problem is to locate sensors to maximize the likelihood that enemy weapons can be intercepted.

These applications sugest that the same terrain may be used repeatedly, with different sets of threats.

Parameters of the Terrain Masking Template

This paragraph explores possible parameterization of the TM problem with a view to using the problem within larger C4I applications and also for different kinds of applications including variations of the given TM problem. Our goal is to explore extensions of the original TM specification into a template with parameters that can be set to obtain variations of the original specification, and where the generalization does not cause inefficiency.

- **Range of threat**. In the given TM problem, the threat range is specified as a distance. Objects farther away from the threat than its range are not visible. Thus, the visibility region of a threat is a circle --- objects outside the circle are not visible to the threat. A more general problem allows the visibility region for a threat to have arbitrary geometry --- a circle is only one possible geometrial structure. Directional antennae, for instance, may be more powerful in one direction than another.
- **Line of sight: The effect of a threat at different points on the terrain**. In the given TM problem, the goal is to compute, for each point in the terrain, the altitude below which the object is not visible, assuming that objects are visible in an unobstructed straight line from the object to the threat. A more general problem is to consider an arbitrary effect for a threat of the following form: the effect of a threat at a point x,y in the grid representing the terrain is a function of the two points on the grid nearest the given point x,y and a straight line joining point x,y to the threat location. This generalization does not make the problem more complex, and the generalization may be useful in variations of the given problem.

  In the given problem, an object is visible from the threat in an unobstructed line, up to the limit of its range; the object is invisible outside the range. Thus, there is an abrupt transition from visibility to invisibility as the object moves out of range. A more realistic model has degrees of visibility, with the object becoming less visible as it moves further away from the location of the threat. The more general model falls within the generalized class of problems: the effect of a threat is computed as it propagates in straight lines away from the threat, and the precise nature of the computation is irrelevant.

- **Minimum masking altitude: the cumulative effect of multiple threats.** In the given TM problem, for each point in the terrain, the minimum visible altitude is computed for each threat; then the minimum of all these altitudes is taken --- below this altitude objects are invisible to all threats. A more general problem is as follows: the "effect" of each threat at each point is computed, and the form of the effect is irrelevant; then the cumulative effect of all the threats at each point is computed --- and we do not care whether the cumulative effect is to take the minimum or carry out a more complex computation. (One of the issues is whether this cumulative effect is associative and commutative --- i.e., can the effect of several threats at a point be computed in arbitrary order?)

  This generalization allows for the consideration of models in which the effect of a threat is a degree of visibility, or the probability that an object is visible from the threat; and, the cumulative effect of visibility from different threats can be a function of the probabilities that the object is visible from each threat. This generalization is achieved without loss of efficiency.

Tailoring the terrain-masking template --- user-supplied functions:
This section is a high-level description of the functions provided by the user to tailor the general terrain-masking template to obtain a specific application. A detailed object-oriented design is provided later. The functions provided by the user fall into two categories: (1) essential or first-level parameters that are necessary to specify the application, and (2) second-level parameters that help in obtaining efficient implementations on target architectures, but are not essential for specifying the problem. Here we give a brief description of the first-level functions.

- **Input**: This function inputs terrain data into processes. The data is partitioned among processes in some manner. The function parameters are the description of the files containing terrain data, and the partitioning of the terrain among processes. (The form of partitioning is a second-level parameter; it need not be specified by the user of the template in which case a default partitioning based on the target architecture is used --- this default scheme can be over-ridden by the user.)
- **Computation of range**: Given a threat, this procedure computes the region of the terrain beyond which the threat cannot "see" objects. For the simple case where the visible region is a circle around the threat, the computation is straightforward. This procedure informs the processes responsible for regions within which the threat can "see" that they must compute the effect of that threat within their regions.
- **Computation of effect:** This procedure takes as input a threat (the threat type and location) and computes the effect of the threat at points in the grid. The effect of a threat is computed along rays (straight lines) emanating from the threat location. The lines extend from the threat location up to the visible extent of the threat. The function used to compute the threat effect is a parameter of the template. A special case of the

function computes the minimum height at each point above which objects are visible, as in the given terrain masking problem.

- **Computation of the cumulative effect of all threats**. User-specified function that is employed to compute the cumulative effect of all the threats at a point, given the effect of each threat at that point. A special case of this function is*min*, which is used in the original terrain masking problem where the effect of a threat is an altitude above which the threat can see objects, and the cumulative effect is the minimum altitude above which some threat can see the object.

A specification of object-oriented templates for terrain masking.
Classes:

- *grid point.* A grid point is a class that defines each point in the terrain.
- *gaming area*: The terrain (the gaming area) is an array of grid points. In the original problem, a data field of grid point is the height of the terrain at that point.
- *threat.* The two important methods for class threat are the methods that (1) define the region of the terrain visible from the threat, and (2) computes the threat effect at a point on the grid given the threat effects at neighboring points on the grid, closer to the threat. See the discussion of computation of range (for method 1), and computation of threat effect (for method 2) in the previous paragraphs.
- *threat effect.* The threat effect is a class which may be as simple as a data type (e.g. altitude) or it could be more complex (e.g. probabilities of visibility at different altitudes).
- *collection of threat effects*. This class is essentially a list of threat effects, and a method that computes the cumulative effect of all the threats in the list.

Secondary methods or pragmas
These methods and data fields help in mapping a program efficiently on a target machine.

- number of processes
- partitioning of terrain among processes
- number of threads within a process

Generalization of the Terrain Masking Problem
In this section we generalize the object definition so that the component classes are more general --- i.e., they are classes that can be used in other problems, and that can be reused.

The concept of a grid occurs widely in many problems, and so we generalize the gaming area to a two-dimensional grid. A grid class (either the base class or a subclass) has the following methods

1. **Process creation for grid management** create processes that manage portions of the grid.
2. **Partition file data among processes**: read data into processes that manage the grid from files.
3. **Broadcase file data**: read data about threats from a file into all processes.

4. **Dataflow computation from a source point**: compute values for each point on a grid given the values at points on the grid closer to a source; in our example, this would be computing the threat effect for a given threat at points in the grid, within the range of the threat, given values of points closer to the source.
5. **Independent computations on all points**: carry out a computation on all points of the grid, with each point independent of the others; in our example, this corresponds to computing the overall threat effect.

What additional methods or classes are needed to convert a grid class into a terrain-masking program? All we need is to compute the range of each threat --- the region outside which the threat has no effect. This computation has a dataflow pattern that does not occur widely.

Design of a User Interface for the Terrain Masking Template

This section explores the design of a user interface that can help in tailoring a terrain masking template to obtain a specific application. Our goal here is exploratory rather than to propose a detailed user interface design; our concern is in evaluating the cost-benefits of templates and keeping a specific user interface in mind is helpful in the evaluation.

Our first suggestion is to design user interfaces for the grid template --- the large class of applications that use grids --- and then specialize the grid template user interface to obtain a terrain masking user interface. The idea once again is reuse, this time the goal is to reuse interfaces.

The user interface for the general grid template should include the following features:

- mechanisms to declare the class of a grid point
- operations to partition the grid among processes
- IO operations to partition files among processes
- IO operations to broadcast information from files among processes
- operations on rows, columns, points, and segments of a grid
- computations with standard patterns of dataflow.

Generalizing the user interface to the terrain masking problem only requires user-interface operations to describe threats, threat location and computation of ranges.

REFERENCES

[Ada 90] Ada 9X Project. *Ada 9X Requirements.* Office of the under secretary of defense for acquisition, Washington, D.C., December 1990.

[Agha 93] Agha, G., P. Wegner, and A. Yonezawa, editors. *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, 1993.

[Amalsi 94] Amalsi, G.S., and A. Gottlieb. *Highly Parallel Computing*, Benjamin/Cummings, second edition, 1994.

[Andrews 83] Andrews, G. and F. Schneider. "Concepts and Notations for Concurrent Programming," *Computing Surveys*, vol 15(1), pages 3-43, March 1983.

[Athas 88] Athas, W. and C.L. Seitz, "Multicomputers: Message-Passing Concurrent Computers," *IEEE Computer*, pages 9-24, August 1988.

[Bagrodia 91] Bagrodia, R. and Shen, C.C., MIDAS: Integrated Design and Simulation of Distributed Systems, *IEEE Transactions on Software Engg,* October, 1991.

[Bagrodia 94] Bagrodia, R. and Liao, W., Maisie: A Language for Design of Efficient Discrete-Event Simulations, *IEEE Transactions on Software Engg,* April, 1994.

[Bagrodia 95] Bagrodia, R., Chandy, M. and Dhagat, M., "UC: A Set-Based Language for Data Parallel Programs", *Journal of Parallel and Distributed Computing*, August 1995

[Balasundaram 91] Balasundaram, V., G. Fox, K. Kennedy, and U. Kremer. "A Static Performance Estimator to Guide Data Partitioning Decisions," *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, 213-223, April 1991.

[Bal 89] Bal, H., J. Steiner, and A. Tanenbaum. "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys*, vol 21(3), pages 261-322, September 1989.

[Banjeree 88] Banjeree, U. "An Introduction to a Formal Theory of Dependence Analysis," *The Journal of Supercomputing*, vol. 2, pages 133-149, 1988.

[Banjeree 93] Banerjee, U., R. Eigenmann, A. Nicolau, and D. Padua. "Automatic Program Parallelization," Proceedings of the IEEE, Vol. 81(2), pages 1-33, February 1993.

[Bell 92] Bell, G. "UltraComputers: A Teraflop Before Its Time," *Communications of the ACM*, Vol. 35(8), pages 27-47, August 1992.

[Bennett 90a] Bennett, J.K., J.B. Carter, and W. Zwaenepoel. "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP),* Sigplan Notices, Vol. 25(3), pages 168-175, March 1990.

[Bennett 90b] Bennett, J.K., J.B. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures , *Proceedings of the 17th Annual International Symposium on Computer Architecture*, IEEE Computer Society Press, pages 125-134, 1990.

[Birman 87] Birman, K. and Joseph, T, "Reliable Communications in the Presence of Failures," *ACM Transactions on Computer Systems*, Vol. 5(1), pages 47-76, February, 1987.

[Birrell 84] Birrell, A. and B. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, Vol. 2(1), pages 39-59, February, 1984.

[Blelloch 92] Blelloch, G. "NESL: A Nested Data-Parallel Language, Version 2.4," Carnegie Mellon University CSD technical report no. CMU-CS-92-103, August 1992.

[Bodin 91] Bodin, F., P. Beckman, D.B. Gannon, S. Narayana, and S. Yang. "Distributed pC++: Basic Ideas for an Object Parallel Language," in *Proceedings of Supercomputing '91*, pages 273-282, 1991.

[Brockschmidt 95] Brockschmidt, K. *Inside OLE*, second edition, Microsoft Press, 1995.

[Callahan 88] Callahan, D. and K. Kennedy. "Compiling Programs for Distributed-Memory Multiprocessors," *The Journal of Supercomputing*, Vol. 2, pages 151-169, 1988.

[Carriero 89] Carriero, N. and D. Gelernter. "How to Write Parallel Programs: A Guide to the Perplexed," *ACM Computing Surveys*, vol 21(3), pages 323-357, September, 1989.

[Carter 91] Carter, J.B., J.K. Bennett, and W. Zwaenepoel. "Implementation and Performance of Munin," *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152-164, 1991.

[Chakravarty 90] Chakravarty, I., M. Kleyn, T.Y.C. Woo, R. Bagrodia, and V. Austel. "UNITY to UC: Case Studies in Parallel Program Construction," Schlumberger Laboratory for Computer Science Technical Report No. TR-90-21, November 1990.

[Chandy 88] Chandy, K.M., and J. Misra. *Parallel Program Design: A Foundation*, Addison-Wesley, 1988.

[Chandy 91] Chandy, K.M. and S. Taylor. *An Introduction to Parallel Programming*, Addison-Wesley, 1991.

[Chandy 93] Chandy, K.M. and C. Kesselman. "CC++: A Declarative Concurrent Object-Oriented Programming Notation," in *Research Directions in Concurrent Object-Oriented* Programming, MIT Press, 1993.

[Chen 88a] Chen, M., Y. Choo, and J. Li. "Compiling Parallel Programs by Optimizing Performance," *The Journal of Supercomputing*, vol. 2, pages 171-207, 1988.

[Chen 88b] Chen, M., Y. Choo, and J. Li. "Crystal: From Functional Description to Efficient Parallel Code," ACM, pages 417-433, 1988.

[Custer 93] Custer, H. *Inside Windows NT*, Microsoft Press, 1993.

[Das 91] Das, R., R. Ponnusamy, J. Saltz, and D. Mavriplis. "Distributed Memory Compiler Methods for Irregular Problems -- Data Copy Reuse and Runtime Partitioning," NASA, ICASE Report no. 91-73, September, 1991.

[DDJ 95] Dr. Dobb's Journal, Special Report. *The Interoperable Objects Revolution: How Component Software is Changing the Way You Program*, Winter 1995.

[Dewar 79] Dewar, R., A. Grand, S. Lui, and J. Schwartz. "Programming by Refinement, as Exemplified by the SETL Representation Sublanguage," *ACM Transactions on Programming Languages and Systems*, vol. 1(1), pages 27-49, July 1979.

[Dubois 88] Dubois, M., C. Scheurich, and F.A. Briggs. "Synchronization, Coherence, and Event Ordering in Multiprocessors," *IEEE Computer*, Vol. 21(2), pages 9-21, February 1988.

[El-Rewini 94] El-Rewini, H., T.G. Lewis, and H.H. Ali. *Task Scheduling in Parallel and Distributed Systems*, Prentice Hall, 1994.

[Feo 90] Feo, J., D. Cann, and R. Oldehoeft. "A Report on the SISAL Language Project," *Journal of Parallel and Distributed Computing*, Vol 12(10), pages 349-366, 1990.

[Ferguson 81] Ferguson, R.. "PROLOG A Step Toward the Ultimate Computer Language," *BYTE*, pages 384-399, November 1981.

[Ferrante 87] Ferrante, J., K.J. Ottenstein, and J.D. Warren. "The Program Dependence Graph and Its Use in Optimization, *ACM Transactions on Programming Languages and Systems*, Vol. 9(3), pages 319-349, July 1987.

[Fortune 78] Fortune, S. and J. Wyllie. "Parallelism in Random Access Machines," *Proceedings of the Tenth ACM Symposium on Theory of Computing*, pages 114-118, 1978.

[Foster 90a] Foster, I. and S. Taylor. *Strand: New Concepts in Parallel Programming*, Prentice Hall, 1990.

[Foster 94c] Foster, I. *Designing and Building Parallel Programs*, Addison-Wesley, 1994.

[Foster 95] Foster, I. and K.M. Chandy. "FORTRAN M: A Language for Modular Parallel Programming," *Journal of Parallel and Distributed Computing*, Vol. 25(1), 1995.

[Fox 90] Fox, G., S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. *FORTRAN D Language Specification*, technical report TR90-141, Rice University department of computer science, 1990.

[Freudenberger 83] Freudenberger, S., J. Schwartz, and M. Sharir. "Experience with the SETL Optimizer," *ACM Transactions on Programming Languages and Systems*, vol. 5(1), pages 26-45, January 1983.

[Gajski 82] Gajski, D., D. Padua, D. Kuck, and R. Kuhn. "A Second Opinion on Data Flow Machines and Languages," *IEEE Computer*, February, 1982, pages 58-69.

[Gamma 95] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Gehani 84] Gehani, N., "Broadcasting Sequential Processes", *IEEE Transactions on Software Engg*, SE-10(4), 1984, pages 343-351.

[Geist 92] Geist, G.A. and V.S. Sunderam. "Network-Based Concurrent Computing on the PVM System," *Concurrency: Practice and Experience*, Vol 4(4), 293-311, June 1992.

[Gharachorloo 90] Gharachorloo, K., D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. "Memory Consistency and Event Ordering in Scaleable Shared-Memory Multiprocessors," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, Seattle WA, IEEE Computer Society Press, pages 15-26, May 1990.

[Gibbons 93] Gibbons, P.B., R.M. Karp, C.E. Leiserson, and G.M. Papadopoulos. *Proceedings of the DIMACS Workshop on Models, Architectures, and Technologies for Parallel Computation*, September 1993.

[Gregory 87] Gregory, S. *Parallel Logic Programming in Parlog*, Addison Wesley, 1987.

[Guzzi 90] Guzzi, M.D., D.A. Padua, J. Hoeflinger, and D.H. Lawrie. "Cedar FORTRAN and Other Vector and Parallel FORTRAN Dialects," *Journal of Supercomputing*, Vol. 4(1), pages 37-62, 1990.

[Hasselbring 93] Hasselbring, W. "Prototyping Parallel Algorithms with ProSet-Linda," in J. Volkert's *Parallel Computation (Proceedings of the Second International ACPC Conference),* Lecture Notes in Computer Science, Vol. 734, Springer-Verlag, pages 135-150, October 1993.

[Hatcher 91b] Hatcher, P., A. Lapadula, R. Jones, M. Quinn, and R. Anderson. "A Production-Quality C* Compiler for Hybercube Multicomputers," *ACM Sigplan*, Vol. 26(7), pages 73-82, 1991.

[Hayesroth 95] Hayesroth, B., K. Pfleger, P. Lalanda, P. Morignot, and M. Balabanovic. "A Domain-Specific Software Architecture for Adaptive Intelligent Systems," *IEEE Transactions on Software Engineering*, Vol. 21(4), pahes 288-301, April 1995.

[Hiranandani 91b] Hiranandani, S., K. Kennedy, C. Koelbel, U. Kremer, and C-W. Tseng. "An Overview of the FORTRAN D Programming System," *Lecture Notes in Computer Science*, 589, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua (Eds.) *Languages and Compilers for Parallel Computing*, pages 18-34, August 1991.

[Hoare 78] Hoare, C.A.R. "Communicating Sequential Processes," *Communications of the ACM*, Vol. 21(8), pages 666-677, August 1978.

[Hoare 84] Hoare, C.A.R. *Communicating Sequential Processes*, Prentice Hall, 1984.

[HOPL 93] *Proceedings of the Second History of Programming Languages Conference*, ACM Sigplan Notices Volume 28, Number 3, March 1993.

[Hudak 89] Hudak, P. "Conception, Evolution, and Application of Functional Programming," *ACM Computing Surveys*, Vol. 21(3), pages 359-411, September 1989.

[Hudak 90] Hudak, P. and P. Wadler (editors). *Report on the Functional Language Haskell*, Technical Report YaleU/DCS/RR-777, Yale University department of computer science, April 1990.

[Inmos 88] Inmos. *Occam 2 Reference Manual*, Prentice Hall, 1988.

[Intel 91] Intel Corporation literature, November 1991.

[King 94] King, D.P. *Investigation into Formalization of Domain-Oriented Parallel Software Development*, Thesis AFIT/GCE/ENG/93D-08, USAF, 1994.

[Kligerman 86] Kligerman, E. and A.D. Stoyenko. "Real-Time Euclid - A Language for Reliable Real-Time Systems," *IEEE Transactions on Software Engineering*, Vol 12(9), pages 941-949, 1986.

[Knuth 76] Knuth, D.E. "Big Omicron, Big Omega, and Big Theta," *SIGACT News (ACM)*, Vol. 8(2), pages 18-24, 1976.

[Koelbel 91] Koelbel, C., and P. Mehrotra. "Compiling Global Name-Space Parallel Loops for Distributed Execution," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2(4), pages 440-451, October 1991.

[Koelbel 94] Koelbel, C., D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance FORTRAN Handbook*, MIT Press, 1994.

[Lamport 79] Lamport, L. "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, pages 690-691, September 1979.

[Lamport 94] Lamport, L. "The Temporal Logic of Actions," *ACM Transactions on Programming Languages and Systems*, Vol 16(3), pages 872-923, May 1994.

[Leighton 92] Leighton, F. Thompson. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes*, Morgan Kaufmann, 1992.

[Leiserson 90] Leiserson, C., Z. Abuhamdeh, D. Douglas, C. Feynman, M. Ganmukhi, J. Hill, W. Hillis, B. Kuszmaul, M. St. Pierre, D. Wells, M. Wong, S. Yan, and R. Zak. "The Network Architecture of the Connection Machine," in *Thinking Machines Corporation technical summary*, 1990.

[Lenoski 92] Lenoski, D., J. Laudon, K. Gharachorloo, W-D. Weber, A. Gupta, J.L. Hennessy, M. Horowitz, and M. Lam. "The Stanford Dash Multiprocessor," *IEEE Computer*, March 1992, pages 63-79.

[Li 89] Li, K. and P. Hudak. "Memory Coherence in Shared Virtual Memory Systems," *ACM Transactions on Computer Systems*, Vol. 7(4), pages 321-359, November 1989.

[Li 91] Li, J.K. and M. Chen. "Compiling Communication Efficient Programs for Massively Parallel Machines," *IEEE Transactions on Parallel and Distributed* Systems, Vol 2(3), pages 361-376, 1991.

[Luqi 93] Luqi. "Real Time Constraints in a Rapid Prototyping Language," *Computer Languages*, Vol. 18(2), pages 77-103, 1993.

[Mehrotra 91] Mehrotra, P. and J. Van Rosendale. "Programming Distributed Memory Architectures Using Kali," in *Advances in Languages and Compilers for Parallel Computing*, MIT Press, 1991.

[Morse 94] Morse, H.S. *Practical Parallel Computing*, AP Professional, 1994.

[MPI 93] Message Passing Interface Forum. "MPI: A Message Passing Interface," *Proceedings of Supercomputing '93*, pages 878-883, IEEE Computer Society, 1993.

[Nikhil 88] Nikhil, R.S. *Id Reference Manual*, CSG Memo 284, MIT Lab for computer science, Cambridge, MA, August 1988.

[OMG 95] Object Management Group. *Common Object Request Broker Architecture (CORBA) Specification*, version 2.0, 1995.

[Ousterhout 82] Ousterhout, J.K. "Scheduling Techniques for Concurrent Systems," *IEEE Proceedings of the Third International Conference on Distributed Computing Systems*, pages 22-30, 1982.

[Padua 86] Padua, D., and M. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Communications of the ACM*, Vol. 29(12), pages 1184-1192, December 1986.

[Patterson 90] Patterson, D.A., and J.L. Hennessy. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1990.

[Patterson 94] Patterson, D.A. "Networks of Workstations," UC Berkeley Technical Report, 1994.

[Peek 93] Peek, J., T. O' Reilly, and M. Loukides. *Unix Power Tools*, O' Reilly and Associates, 1993.

[Polychronopoulos 88] Polychronopoulos, C. "Toward Auto-scheduling Compilers," *The Journal of Supercomputing*, Vol. 2, pages 297-330, 1988.

[POSIX 90] POSIX P1003.4a. "Threads Extension for Portable Operating System," IEEE, 1990.

[Powell 94] Powell, M.L., S.R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. "Sun OS Multi-thread Architecture", Sun Microsystems White Paper, 1994.

[Quinn 94] Quinn, M. *Parallel Computing: Theory and Practice*, McGraw Hill, 1994.

[Rashid 86] Rashid, R.F. "Threads of a New System," *Unix Review*, Volume 4, pages 37-49, August 1986.

[Reich 95] Reich, D.E. *Designing High-Powered OS/2 Warp Applications: The Anatomy of Multithreaded Programs*, John Wiley and Sons, 1995.

[Richter 94] Richter, J.M. *Advanced Windows NT*, Microsoft Press, 1994.

[Rine 95] Rine, D.C. *Object-Oriented Systems and Applications*, IEEE Computer Society Press, 1995.

[Rosing 91b] Rosing, M., R. Schnabel, and R. Weaver. "The DINO Parallel Programming Language," *Journal of Parallel and Distributed Computing*, Vol. 13, pages 30-42, 1991.

[Saltz 90] Saltz, J., K. Crowley, R. Mirchandaney, and H. Berryman. "Run-Time Scheduling and Execution of Loops on Message Passing Machines," *Journal of Parallel and Distributed Computing*, Vol. 8, pages 303-312, 1990.

[Saltz 91] Saltz, J., H. Berryman, and J. Wu. "Multiprocessors and run-time Compilation," *Concurrency: Practice and Experience*, Vol. 3(6), pages 573-592, December 1991.

[Saraswat 87a] Saraswat, V., "GHC: Operational Semantics, Problems, and Relationship with CP," *Proceedings of the Symposium on Logic Programming*, IEEE Computer Society, pages 347-358, 1987.

[Seitz 85] Seitz, C.L. "The Cosmic Cube," *Communications of the ACM*, Vol. 28(1), pages 22-33, January 1985.

[Seitz 90] Seitz, C.L. "Multicomputers," in C.A.R. Hoare, editor, *Developments in Concurrency and Communication*, pages 131-200, Addison-Wesley, 1990.

[Shapiro 89] Shapiro, E. "The Family of Concurrent Logic Programming Languages," Department of Applied Math and Computer Science, The Weizmann Institute of Science, Israel, technical report no. CS89-08, May 1989.

[Shaw 89] Shaw, A. "Reasoning about Time in Higher Level Language Software", *IEEE Transactions on Software Engg*, SE-13(7), July, 1989.

[Sipelstein 91] Sipelstein, J.M. and G.E. Blelloch, "Collection Oriented Languages," *Proceedings of the IEEE*, Vol. 79(4), pages 504-523, 1991.

[SOM 94] *SOMobjects: A Practical Introduction to SOM and DSOM*, Document Number GG24-4357-00, IBM International Technical Support Organization, Austin Center, July 1994.

[Stagray 95] Stagray, K., and L.S. Rogers. *Official Guide to Using OS/2 Warp*, IBM Press, 1995.

[Sunderam 90] Sunderam, V. "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice and Experience*, Vol 2(4), pages 315-339, December 1990.

[Tanaka 86] Tanaka, J., K. Ueda, T. Miyazaki, A. Takeuchi, Y. Matsumoto, and K. Furukawa, "Guarded Horn Clauses and Experiences with Parallel Logic Programming," ICOT Technical Report, no. TR-168, April 1986.

[Tanenbaum 87] Tanenbaum, A.S. *Operating Systems Design and Implementation*, Prentice-Hall, 1987.

[Tanenbaum 89] Tanenbaum, A.S. *Computer Networks*, second edition, Prentice-Hall, 1989.

[Tanenbaum 95] Tanenbaum, A.S. *Distributed Operating Systems*, Prentice-Hall, 1995.

[Thinking 90] Thinking Machines Corporation. *The Connection Machine CM-2 Technical Summary*, 1990.

[Thinking 91] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary*, 1991.

[van de Snepscheut 93] van de Snepscheut, J.L.A. *What Computing is All About*, Springer-Verlag, 1993.

[Walinsky 94] Walinsky, C. and D. Banerjee, "A Data Parallel FP Compiler," *Journal of Parallel and Distributed Computing*, Vol. 22(2), pages 138-153, August 1994.

[Warren 87] Warren, D. and L. Pereira. "Prolog- The Language and its implementation Compared with LISP," *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, Sigart Notices, Vol. 12(8), August 1987; Sigart Newsletter, No. 64, pages 109-115, August, 1987.

[Wegner 83] Wegner, P. and S. Smolka. "Processes, Tasks, and Monitors: A Comparitive Study of Concurrent Programming Primitives," *IEEE Transactions on Software Engineering*, Vol. SE-9(4), pages 446-462, July 1983.

[Zhao 87] Zhao, W., K. Ramamritham, and J.A. Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems," *IEEE Transactions on Software Engineering*, Vol. 13(5), pages 564-577, 1987.

[Zima 88] Zima, H.P., H.J. Bast, and M. Gerndt. "SUPERB: A Tool for Semi-Automatic MIMD\SIMD Parallelization," *Parallel Computing*, Vol. 6, pages 1-18, 1988.

[Zima 91] Zima, H.P. and B. Chapman. *Supercompilers for Parallel and Vector Computers*, Addison Wesley, 1991.

[Zorn 89] Zorn, B., K. Ho, J. Larus, L. Semenzato, P. Hilfinger, "Lisp Extensions for Multiprocessing," Proceedings of the 22nd Hawaii International Conference on System Sciences, January 1989.

☆U.S. GOVERNMENT PRINTING OFFICE: 1998-610-130-61144